# Table of Contents

Table of Contents........................................................................................................... 1  
Preface............................................................................................................................. 3  
List of Participants......................................................................................................... 5  
Presentations.................................................................................................................. 7  
A Qualitative Comparison of three Aspect-Mining Techniques (Kim Mens)................. 9  
Metadata and Aspect Evolution (Bram Adams)........................................................... 15  
Effects of Defects in UML Models (Christian Lange)................................................ 23  
Assessing the Correspondence between Design and Implementation (Dennis van Opzeeland).............................................................................................................37  
MetricView Evolution (Martijn Wijns)........................................................................ 47  
Heuristics based on Reconstruction (Roel Wuyts)....................................................... 57  
Challenges in Software Evolution (Tom Mens)............................................................. 61  
Software Evolution Case Studies (Filip van Rysselberghe)........................................... 71  
Analysing Refactorings with Graph-transformation Techniques (Tom Mens).............. 79  
Refactoring Architectural Style (Marc van Kempen)..................................................... 89  
CVSscan – Visualization of Software Evolution (Lucian Voinea)................................. 101  
Bad Smells and Refactorings (Peter Ebraert)................................................................ 111  
Understanding Change – where do we look at? (Filip van Rysselberghe)............... 115
Preface

BENEVOL is a series of workshops for researches from Belgium and the Netherlands in the domain of software evolution. The BENEVOL workshop is a platform for researchers to present finished and ongoing research and to discuss with their colleagues. The goal of BENEVOL is to stimulate collaboration between the workshop participants. Previous editions of BENEVOL were held at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam (2003) and at the Universiteit Antwerpen (2004).

The third edition of BENEVOL was held at the Technische Universiteit Eindhoven (TU/e) on May 26th and May 27th, 2005. Researches from various institutions in Belgium and the Netherlands participated in the workshop. The workshop was organized by Christian Lange and Michel Chaudron from the System Architecture and Networking group (SAN) at the TU/e and Tom Tourwé from the CWI.

The program of the workshop contained 14 presentations that were grouped into sessions covering the following topics:

- Aspect-Oriented Software Evolution, including evolution of aspect programs, identification of aspects, and extraction of aspects. (Session chair: Tom Tourwé)
- Model-Driven Software Evolution, including model extraction, code generation, and co-evolution. (Session chair: Michel Chaudron)
- Formal Foundations of Software Evolution, including formal refactorings, and models for evolution. (Session chair: Tom Verhoeff)
- Understanding Evolution, including quality metrics, visualizing evolution, and studying change histories. (Session chair: Kim Mens)
- Tool demonstrations.

This report contains the slides of twelve of the presentations. Most presentations were revised after the workshop such that they cover the feedback from the discussions at BENEVOL. Contact information of the authors is provided in the presentations and additional information is available on the authors’ websites.

We would like to thank all participants of the third edition of BENEVOL for their attendance, presentations and discussions, and Cecile Brouwer and Richard Verhoeven for their help during the organization, which made it a successful workshop. We would also like to thank the System Architecture and Networking group of the TU/e and the FWO WOG scientific research network on Foundations of Software Evolution for sponsoring the workshop.

Eindhoven, January 2006

Christian F. J. Lange
Michel R.V. Chaudron
Tom Tourwé
Participants

Bram Adams, Universiteit Gent
Michel Chaudron, Technische Universiteit Eindhoven
Serge Demeyer, Universiteit Antwerpen
Peter Ebraert, Vrije Universiteit Brussel
Andy Kellens, Vrije Universiteit Brussel
Marc van Kempen, Technische Universiteit Eindhoven
Christian Lange, Technische Universiteit Eindhoven
Kim Mens, Université catholique de Louvain
Tom Mens, Université de Mons-Hainaut
Dennis van Opzeeland, Technische Universiteit Eindhoven
Reinier Post, Technische Universiteit Eindhoven
Coen De Roover, Vrije Universiteit Brussel
Filip Van Rysselberghe, Universiteit Antwerpen
Hans Schippers, Universiteit Antwerpen
Tom Tourwe, Centrum voor Wiskunde en Informatica
Tom Verhoeff, Technische Universiteit Eindhoven
Lucian Voinea, Technische Universiteit Eindhoven
Martijn Wijns, Technische Universiteit Eindhoven
Roel Wuyts, Université libre de Bruxelles
Presentations
A Qualitative Comparison of Three Aspect Mining Techniques

M. Ceccato, M. Marin, K. Mens, L. Moonen, T. Tourwé, P. Tonella

Aspect mining

- Identification of crosscutting concerns in existing software systems
- Starting point for system exploration
- Support program comprehension, software maintenance and evolution
  - e.g. migrating to an AOP solution

Goal: high degree of automation
Comparison and combination of mining techniques

• Understand what “assumptions” (about crosscutting concerns) the techniques rely on
• Evaluate strengths and weaknesses
• Mutual filtering / completion
• Enhance automation through a multi-technique approach and tool

Three aspect mining techniques

– Identifier Analysis –

Use FCA to group classes/methods with similar names
Three aspect mining techniques
– Dynamic Analysis –

Use FCA to associate methods with the most specific use case scenarios in which they are executed

<table>
<thead>
<tr>
<th>scen 1</th>
<th>scen 2</th>
<th>scen 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>meth_1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>meth_2</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>meth_3</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Concept lattice with sparse labeling

Three aspect mining techniques
– Fan-in analysis –

Concerns
- Contract enforcement
- Consistent behavior
- Scattered implementation relying on common functionality
- Design patterns with specific structure
JHotDraw
– common benchmark for aspect mining –

- Framework for 2D graphics
- ~18,000NCLOC
- Open-source (jhotdraw.org)
- Good design – GoF patterns (Gamma et al.)

Comparison

<table>
<thead>
<tr>
<th>Concern</th>
<th>Fan-in analysis</th>
<th>Identifier Analysis</th>
<th>Dynamic Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Consistent Behavior / Contract Enforcement</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Command Execution</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Bring to front / Send to back</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Manage Handles</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Move Figures</td>
<td>+ (discarded)</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Conclusions drawn from the results

• Limitations
  – **Dynamic** analysis: misses functionalities exercised by *all* traces
  – **Fan-in**: only crosscutting with large extent
  – **Identifier** analysis: relies on naming conventions

• Combination (orthogonal properties) – enhance automation and improve individual results

Combination of techniques

• **Increased coverage**
  - the union of discovered results (fan-in + dynamic)

• **Improved completeness** for the discovered aspect “seeds”
  - more elements relevant to the aspect (+ identifier)

• **Coarse-grained aspects**
  - grouping of identifier analysis concepts (fan-in/dynamic)

• **Filtering**
  - Discard irrelevant concepts
Resources

• Detailed results
  – **Fan-in**: swerl.tudelft.nl/amr
  – **Dynamic analysis**: star.itc.it/dynamo/jhotdraw-detailed-results.html
  – **Identifier analysis**: ask me 😊
• *JHotDraw* as benchmark and *AJHotDraw* as showcase for aspect refactoring
• Tools: *Dynamo, FanInTool, DelfSTof*
• Collaborations
  – *AIRCo/AIRPort*
Metadata and aspect evolution

Experiences in Aspicere

Bram ADAMS
Software Engineering Lab, INTEC, UGent

Aspicere

• What’s in a name?
  • aspicere ≡ “to look at” (Latin)
  • Here: aspect language for C

• Characteristics:
  • Prolog-based pointcut language
  • Source code weaver
  • Currently only statically determinable joinpoints
  • Likewise no weaving within advices

• Future:
  • Merging into GCC 4.0 (“heterogeneous AOP")
  • cflow, sequence, ...
  • Weaving inside advices
Outline

1. Aspicere, a short introduction

General architecture

- Weaver ≡ Source-to-source transformer
- Source-to-source transformer for GCC
More details

1. Parser:
   - btyacc (backtracking): slowwww...
   - Antlr: very fast + type-checking

2. Extraction:
   - XSLT + XPath (cached)

3. Joinpoint matching (Prolog):
   - Backward chaining
   - Instantiate joinpoints as needed
   - Bind weave-time properties

4. Weaving:
   - Depends on joinpoint type
   - Highly procedural

5. De-XMLify:
   - XML to source code

Even more details ...

```
int f(int* a, double b);
int main(void){
  ...
  res=f(ptr, 5.0);
  ...
}

int f_caller_proxy(int* a, double b){
  ...
}

void log(thisJoinPoint* jp){
  ...
}

int f_callee_proxy(thisJoinPoint* jp){
  ...
}

int advice log() on(Jp):
  ... { ... }
```
Example

ReturnType advice tracing_nonvoid(ReturnType) on (Jp):
    call(Jp,_)
    && type(Jp,ReturnType)
    && !str_matches("void",ReturnType)
    {
        ReturnType i;
        /* Tracing code */
        i = proceed();
        /* Tracing code */
        return i;
    }

Aspect ≡ normal compilation unit enhanced with advice

Bindings

• What?
  • Logic variables which are bound and can be used freely throughout advice code
  • ≈ C++ template parameter
  • cf. Kris Gybels’ and Johan Brichau’s work, Cobble, LogicAJ, ...

• How?
  • Consider tuple of bindings L=(L₁,...,Lₙ)
  • Instantiate advice once for all solutions for L

Why?
  • Leverage power of Prolog ⇒ reusable, robust pointcuts
  • NECESSITY ⇒ no Object-class, nor template parameters

generic aspect language
Outline

2. Metadata

Metadata

• What?
  • “data about data”: semantics, design decisions, conventions, ...
• Why?
  • automated (aspectized) evolution, aspect mining, ...
• How?
  • Documentation ➔ Javadoc, Doxygen, ...
  • Separate file ➔ property files, ...
  • Language support ➔ Java 5 annotations, C# custom attributes
  • AOP introduction ➔ AspectJ 5
• In Aspicere:
  • Prolog facts & rules ≡ ...
• Future:
  • What about annotations in C?

- loose coupling
- no fixed metadata source
- delocalized
Metadata supply and consumption

Outline

3. Demonstration
Conclusion and questions

• Conclusion:
  • Prolog facts and rules enable transparent storing of metadata
  • Aspicere’s use of Prolog-like pointcuts allows easy exploitation of metadata

• Questions:
  • Does direct language support for metadata (a.k.a. annotations) yield better evolution opportunities than other mechanisms?
  • What about availability of metadata?

References:
• Laddad, R. (2005): “AOP and metadata: A perfect match, Part 1 and 2” (IBM developerWorks)
Effects of Defects in UML Models

Christian Lange
C.F.J.Lange@TUE.nl
Michel Chaudron
M.R.V.Chaudron@TUE.nl
www.win.tue.nl/~clange
www.win.tue.nl/empanada

BENEVOL workshop, May 26th 2005

Overview

- Introduction and Motivation
- Experimental Design
- Results
- Analysis
- Conclusion
Introduction: Empanada

- EmpAnADa project
  - Empirical Analysis of Architecture and Design Quality
    - Early evaluation of quality attributes
    - Metrics for Architecture and Design models
  - Improvement by Refactoring
  - Completeness and Consistency checking
  - Development of methods
  - Method validation by Empirical Studies
    - Direct feedback
    - Exploring problems in practice

Introduction: UML issues

- Unified Modeling Language
  - No formal semantics
  - Offers large degree of freedom
    - Extension Mechanisms
    - 9 Diagram types (UML 2 has even 13!!)
      - Diagrams are overlapping!
  - UML is used in many different ways
  - UML Defects:
    - Completeness
    - Consistency
Introduction: Defects

- Correct
  - Matching elements in overlapping parts of diagrams

- Completeness defect
  - Missing element

- Consistency defect
  - Mismatch
  - Conflicting elements

Detection by checking rules
- SAAT
- RPA

Case Study Results

Our case studies have shown large numbers of defects

How serious are defects?

<table>
<thead>
<tr>
<th>Consistency</th>
<th>A1</th>
<th>A2</th>
<th>B1</th>
<th>B2</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages without Name</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.28%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.46%</td>
</tr>
<tr>
<td>Messages without Method</td>
<td>58.73%</td>
<td>7.62%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>27.14%</td>
<td>27.40%</td>
</tr>
<tr>
<td>Messages between unrelated Classes</td>
<td>71.94%</td>
<td>79.37%</td>
<td>77.73%</td>
<td>43.14%</td>
<td>81.90%</td>
<td>81.74%</td>
</tr>
</tbody>
</table>
Purpose of Study

- Defects occur in practice, but does it matter?
  - Are defects detected in practice?
  - If not, do defects lead to misunderstandings?

- Goal (GQM)
  - Analyze defects in UML models for the purpose of identifying issues and observations with respect to misunderstandings and ambiguities from the perspective of the researcher in the context of TUE grad students and professionals.

Experiment Design

- Give UML models to subjects
- Ask question about UML model
  - Two types
    - Which implementation matches the diagrams?
    - How do you interpret these diagrams?
  - Answer from the perspective of the developer.

- Multiple-choice test
  - 4 options
  - + 1 option ("There's something wrong, can't give an answer")

- Background questions

- Pilot run was performed
Experimental Treatment

- **Treatment**
  - one of 9 defects
  - no defect

- **Defects are injected in model**
  - Each *infected* question is paired with a *control question*

- **All subjects receive all treatment levels**
  - "same subject design"
  - by definition “balanced”
    - all treatment groups have the same background

Treatments: Defects

- Message without Name
- Message Name does not correspond to Method
- Message in the wrong direction
- Class from CD not in SD
- Class from SD not in CD

- Use Case without SD
- Multiple definitions of Class with equal Name
- Method from SD not in CD
- Method from CD not in SD
Example Question

Suppose you are developer in this banking software project. It is your task to implement class ATM. Please indicate how you would implement the ATM class given these two UML diagrams?

A) Class ATM{
Method
getCardInserted()
{{
c.requestPIN();
dosomething;
a.open()
}}
...
}

B) Class ATM{
Method
getCardInserted()
{{
c.requestPIN();
dosomething;
a.lock()
}}
...
}

C) Class ATM{
Method
getCardInserted()
{{
c.requestPIN();
dosomething;
a.acknowledge()
}}
...
}

D) Class ATM{
Method
getCardInserted()
{{
c.requestPIN();
dosomething;
a.validate()
}}
...
}

Something is wrong!

Results of Question

Inconsistency:
Message Name does not correspond to Method
Subjects

- 110 Students
  - 1st year MSc programme (TUE)
  - Course: Software Architecture (2II10)
  - Preparation
    - Lecture about UML
    - 5 weeks assignment: designing and evaluating UML

- Professionals
  - Completed online questionnaire
  - Emailed URL to contacts and newsgroups
  - 45 answered Q1 - 24 answered Q10
Subjects’ Experience II

Students

- Degree:
  - 85%
  - 10%
  - 5%
- BSc CS
- BSc Electrical Engineering
- Other

Professionals

- Years of working experience:
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8
  - More

Results: Detection

- Detection rate:
  - Defected control
  - Type of question
  - Control
Results: Detection II

- Defects
  - Sorted by detection rate (students)

<table>
<thead>
<tr>
<th>Defect</th>
<th>Stud.</th>
<th>Prof.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class not in SD (symb.)</td>
<td>0,95</td>
<td>0,96</td>
</tr>
<tr>
<td>Message without Name</td>
<td>0,69</td>
<td>0,60</td>
</tr>
<tr>
<td>Message in the wrong Direction</td>
<td>0,60</td>
<td>0,58</td>
</tr>
<tr>
<td>UC without SD</td>
<td>0,50</td>
<td>0,52</td>
</tr>
<tr>
<td>Method not in CD</td>
<td>0,49</td>
<td>na</td>
</tr>
<tr>
<td>Message without Method (symb.)</td>
<td>0,49</td>
<td>0,33</td>
</tr>
<tr>
<td>Class not in SD</td>
<td>0,47</td>
<td>0,68</td>
</tr>
<tr>
<td>Message without Method</td>
<td>0,39</td>
<td>0,38</td>
</tr>
<tr>
<td>Class not in CD</td>
<td>0,18</td>
<td>0,11</td>
</tr>
<tr>
<td>Method not in SD</td>
<td>0,14</td>
<td>na</td>
</tr>
<tr>
<td>Multiple Class defs.</td>
<td>0,10</td>
<td>0,33</td>
</tr>
</tbody>
</table>

- **Average** 0,46 0,50
- **Std Dev** 0,26 0,25
- **Max** 0,95 0,96
- **Min** 0,10 0,11

Variability Measure (Entropy)

- If a defect is not detected, does it lead to misinterpretation?

\[
VarM(k_0, ..., k_{i-1}) = 1 - 2 \frac{\sum_{i=0}^{K} k_i}{N(K-1)}
\]

- K = Number of options, N = Sum of all answers,
- \(k_i\) = # of answers of option i (\(k_i\)'s are ordered: \(k_i \leq k_{i+1}\))

- Intuition: Measuring how discriminating a distribution is

VarM=0

< VarM=1
Results: Variability

- Variability
  - Defect Control
  - 0.00
  - 0.20
  - 0.40
  - 0.60
  - 0.80
  - 1.00

- Sorted according to VarM

<table>
<thead>
<tr>
<th>Defect</th>
<th>Stud.</th>
<th>Prof.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Class def. (meth.)</td>
<td>0.92</td>
<td>0.68</td>
</tr>
<tr>
<td>Message without Method (symb.)</td>
<td>0.86</td>
<td>0.94</td>
</tr>
<tr>
<td>Message without Method</td>
<td>0.84</td>
<td>0.90</td>
</tr>
<tr>
<td>UC without SD</td>
<td>0.83</td>
<td>0.44</td>
</tr>
<tr>
<td>Class not in CD (meth.)</td>
<td>0.83</td>
<td>0.93</td>
</tr>
<tr>
<td>Method not in CD</td>
<td>0.69</td>
<td>n/a</td>
</tr>
<tr>
<td>Method not in SD</td>
<td>0.67</td>
<td>n/a</td>
</tr>
<tr>
<td>Class not in SD</td>
<td>0.49</td>
<td>0.64</td>
</tr>
<tr>
<td>Message in the wrong Direction</td>
<td>0.47</td>
<td>0.95</td>
</tr>
<tr>
<td>Message without Name</td>
<td>0.47</td>
<td>0.44</td>
</tr>
<tr>
<td>Class not in SD (symb.)</td>
<td>0.34</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Average: 0.71, 0.80
Std Dev: 0.16, 0.21
Max: 0.86, 0.95
Min: 0.47, 0.44
### Severity

- **Product of**
  - Detection rate
  - VarM

- **Combination of** low detection rate and many different interpretations (low VarM) → causes most misunderstandings

<table>
<thead>
<tr>
<th>Defect</th>
<th>Stud.</th>
<th>Prof.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message without Name</td>
<td>0,37</td>
<td>0,26</td>
</tr>
<tr>
<td>Message in the wrong Direction</td>
<td>0,32</td>
<td>0,55</td>
</tr>
<tr>
<td>Class not in SD (symb.)</td>
<td>0,32</td>
<td>0,14</td>
</tr>
<tr>
<td>Class not in SD</td>
<td>0,24</td>
<td>0,44</td>
</tr>
<tr>
<td>Method not in CD</td>
<td>0,15</td>
<td>n/a</td>
</tr>
<tr>
<td>UC without SD</td>
<td>0,09</td>
<td>0,23</td>
</tr>
<tr>
<td>Message without Method (symb.)</td>
<td>0,07</td>
<td>0,31</td>
</tr>
<tr>
<td>Message without Method</td>
<td>0,06</td>
<td>0,34</td>
</tr>
<tr>
<td>Method not in SD</td>
<td>0,05</td>
<td>n/a</td>
</tr>
<tr>
<td>Class not in CD</td>
<td>0,03</td>
<td>0,21</td>
</tr>
<tr>
<td>Multiple Class defs.</td>
<td>0,01</td>
<td>0,23</td>
</tr>
</tbody>
</table>

### Domain Knowledge

“There are no defects in UML models that IPA members create!”

- Can defects be corrected by using context knowledge (domain knowledge)?

- We made equal questions
  - One version with “context” (Traincrossing, Sensor)
  - One version without (Class A, Method 3)

(Cultural background was taken into account)
**Domain Knowledge II**

<table>
<thead>
<tr>
<th></th>
<th>Domain Knowledge</th>
<th>No Domain Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection Rate</td>
<td>0.47</td>
<td>0.95</td>
</tr>
<tr>
<td>VarM</td>
<td>0.49</td>
<td>0.34</td>
</tr>
</tbody>
</table>

- Classes from CD not in SD (Q 5.2)
- Classes from CD not in SD - Controle (Q5.1)
- Classes from CD not in SD (symbolic) (Q 9.2)
- Classes from CD not in SD (symbolic) - Controle (Q9.1)

**Generalizability**

- Is it valid to generalize the outcome of the students experiment?
  - Pearson Correlation
    - Between student results and professional results
      - For detection rate: 0.929 (p-value < 0.001)
      - For VarM: 0.643 (p-value < 0.004)
    - “significant” = p-value < 0.05
Order of Diagrams

- In most cases the sequence diagram was regarded as the *leading* diagram
- To investigate order effects, we presented some questions in a second run with the diagrams in reversed order

Pearson Correlation

- **Professionals vs. reversed**
  - Detection rate: 0.824 (p-value: 0.044)
  - VarM: 0.899 (p-value: 0.015)

- **Students vs. reversed**
  - Detection rate: 0.913 (p-value: 0.011)
  - VarM: 0.638 (p-value: 0.173)

Conclusions

- It makes sense to detect defects!
- Defects in UML models
  - are detected only to a rather low degree
  - do cause misunderstandings
- We ordered defects according to severity
- Domain knowledge matters!
  - But can cause misunderstandings
- Order of models does not matter

Fault-injection in UML models
Lesson learned

- Experimenting is not as simple as it seems
  - Design
    - What is the question? Hypothesis?
    - What are the variables?
    - How to distribute the treatments over the subjects?
  - Preparation
    - Experiment Material
  - Carefully instructing the subjects
  - Analysis
    - Using the proper methods
Assessing Correspondence between Design and Implementation

Dennis van Opzeeland
d.j.a.v.opzeeland@student.tue.nl

Eindhoven University of Technology,
The Netherlands

Outline

• Introduction
• What is correspondence?
• Matching of implementation pieces to design elements
• Highlighting differences
• Case study
• Conclusion
Introduction

- Correspondence:
  - Similarity between design and implementation
- Correspondence vs. evolution
  - Correspondence degrades if implementation evolves but design doesn’t
  - Correspondence $\downarrow$
    $\Rightarrow$ Maintainability $\downarrow$
    $\Rightarrow$ Evolution effort $\uparrow$

What is correspondence?

- Expressed in terms of the model elements
  - Design: classes, interfaces, ...
  - Implementation: class declaration, interface specification,....
- Mapping between design elements and implementation elements
- Correspondence system =
  $\sum_{d \in D, i \in I} \text{sim}(d, i)$
Typical deviations from design

- Structural
  - Easy to check
  - Examples
    - Introduction of new classifiers
    - Differences in names
    - Introduction of new operations and attributes
    - Introduction of dependencies and associations
- Behavioral
  - Hard to check
  - Examples
    - Incompatible message sequences
- Not all deviations are equally problematic

Finding the matching

- Given:
  - Set of design classifiers
  - Set of implementation classifiers
- Problem:
  - Find the design pieces and implementation pieces that were meant to be “the same”
- Different approaches
  - Classifier names
  - Structural properties
  - Package information
  - Metric profile
Using package information

- Heuristic:
  - Existing relations between two packages predict other relations
- Requirements
  - Development view in design
  - Directory layout for source code
  - Partial matching exists
- Purpose
  - Limit search space of other methods

Matching with Metric profiles (1)

- There exist correlations between design metrics and implementation metrics of a system
- Correlating metrics define metric profile of a class
  - Let $c$ be a class, then $m(c)=(m_{1,c}, \ldots, m_{n,c})$
  - Pairwise correlations between metrics in design profile and implementation profile
Matching with Metric Profiles (2)

- Let $d$ be a design class and $i$ an implementation class
- Given metric value for design predict value for implementation metric and compare with real value
  \[
  \text{sim}(d,i) = \sum_{x} \rho_{x} | \beta_{0,x} + m(d) \beta_{1,x} - m(i)_{x} |
  \]
- The implementation class that fits best matches to the design class

Case study

- Characteristics
  - Industrial case
    - Firmware for DVD recorder
  - Design
    - UML 1.4
    - 346 classes
  - Implementation
    - C++
    - 777 classes
    - Lines of Code: 2,558,216
- Approach:
  - Initial matching based on names
  - Empirical analysis for metric profile approach
Correlating metrics

<table>
<thead>
<tr>
<th>Design</th>
<th>Implementation</th>
<th>Corr. Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td># Ops. inherited</td>
<td># Ops. inherited</td>
<td>0.924</td>
</tr>
<tr>
<td>Depth of inh. tree</td>
<td>Depth of Inh. tree</td>
<td>0.883</td>
</tr>
<tr>
<td>Coupl. objects</td>
<td>Data abstr. coupl.</td>
<td>0.816</td>
</tr>
<tr>
<td># Ops. inherited</td>
<td># Protected ops.</td>
<td>0.889</td>
</tr>
<tr>
<td># Ops. inherited</td>
<td>Depth of inh. tree</td>
<td>0.829</td>
</tr>
<tr>
<td># Priv. operations</td>
<td># Priv. operations</td>
<td>0.223</td>
</tr>
<tr>
<td># Attributes</td>
<td># Attributes</td>
<td>0.184</td>
</tr>
</tbody>
</table>

For all correlation coefficient measures, the significance level $p < 0.01$

Case study results

- Classification of deviations from design found
  - Introduction of (private/protected) attributes and operations
  - Introduction of new classes (decomposition of design classes)
  - Unused dependencies
  - Changes in inheritance tree
Conclusions

- Matching approaches
  - Matching based on names:
    - 77% of design matched
    - ?% of implementation matched
  - Matching based on Metric Profiles
    - 0% of design matched
    - 0% of implementation matched
  - Metric Profile useful for highlighting deviations
Visualization of differences

- Given a mapping, finding differences is quite straightforward
- Visualization using MetricView
- Overlay diagrams
Further work

- What can be done to prevent correspondence issues?
- How can correspondence be established?
- What is the impact of correspondence issues?
- How much correspondence is needed?
- What about clustering
MetricView Evolution
- Monitoring Architectural Quality -

Martijn Wijns
M.A.M.Wijns@student.tue.nl

Within EmpAnADA project
Supervisors: Michel Chaudron & Christian Lange
Technische Universiteit Eindhoven

Outline
- Motivation & Context
- User Profiles
- Goals & Existing Tooling
- Prototyped Ideas
- Tool Demo
- Findings
- Future Work
Motivation & Context

- **Motivation**
  - Metrics offer benefits but are hard to use
  - Metrics are often separated from model
  - Metrics offer a low abstraction level
  - Current Metric tooling focuses on single point in time

- **Context**
  - **Data:**
    - UML/XMI standard, many dialects
    - External Metrics
    - Matching Data
  - **Tooling:**
    - CASE (Rose, Together, EA, …)
    - Reverse Engineering (Columbus, …)
    - Version Control (SVN, CVS, …)
    - Correspondence (DICT, …)
    - Analysis (SAAT, SDMetrics, …)
User Profiles

- Three types of user profiles
  - **Software Architect**
    - Monitor Quality over time
    - Identify Weak Points
  - **Project Manager / Client**
    - Monitor Quality from a high level perspective
  - **Maintainer**
    - Identify weak points with their cause
    - Estimate impact of needed changes

Goals & Existing Tooling

- **Main Goals**
  - **Monitor Quality**
    - Support analysis for multiple versions
  - **Provide Abstraction from Metrics**
    - Aggregation on Multiple levels
    - Tailorable, using Custom Quality Model
  - **Provide Easy access to Metrics**
    - Fast and Accurate Navigation
    - Short feedback cycle
      - Keep Model and Metrics together
      - One tool to extract and visualize them
Goals & Existing Tooling

- Existing UML Metrics Tooling
  - SAAT
    - Easy Metric Definition: Relational Database / SQL
    - Maintainability (Dialects...): XSLT / Perl parser
    - No Visual feedback
    - Hard to install: Dependencies
  - SDMetrics
    - Fast
    - Many metric definitions / Design rules
    - Visualization rather basic
  - MetricView
    - Integrated Model and Metrics
    - Visualization
    - No integrated Metric calculation

Prototyped Ideas

- Layout-Adjustment
- Correspondence Visualization
- Metrics over time
- Quality Tree
- Timeline
- Lange-Diagram
- Context Diagram
- Search & Highlight
Prototype - Correspondence

- Quickly spot differences
- Layout construction

Secondary goal: Explore MetricView implementation

Prototype – Metrics over Time

- Metrics changing over time
- Changes in structure less clear in case of many versions
Quality Tree

- Tailorable
- Aggregate metrics
- Single point in time

Timeline

- Visualizes the change of Metric or Quality Attribute values over Time
- Also usable for higher abstraction levels
- Does not scale well to multiple attributes
Lange-Diagram

- System Overview, including inter-diagram relations
- Layout Scalability

Context Diagram

- Context in complete model, not just one diagram
Tool Demo

MetricView Evolution

- Benefits:
  - Overview of relations between diagrams
  - Context-Diagram reveals obscured problems
  - Highlighting same element in multiple diagrams
  - Search by keyword for quick navigation

- Disadvantage
  - Class level metrics give too much detail
  - Scalability for large models
Findings

- Suggestions for usage
  - Pattern Detection (context diagram)
  - Impact Prediction (context diagram)
  - Implicit Relations (search & highlight)
- Incomplete models
  - Design Decision not model everything

Future Work

- Improve Visualizations
  - Higher Abstraction levels
  - Reports
  - Dynamic Layout
  - Design Smells
- Multiple Version support
  - Specific Metrics?
  - Integrate with CVS-like tools
- Validation
  - Integration
  - During development, rather than afterwards
  - Usability Testing
Introduction to Software Metrics

- What are software metrics?
  - Specification for Quantification of Software Attributes
- What can you do with them?
  - Understand
  - Control
  - Improve
- Who wants them?
  - Anyone who wants to systematically understand, control or improve software quality
- Why Architecture Metrics?
  - Early detection means easier/cheaper to fix
Type Reconstruction

Context: program understanding in dynamically typed languages
- e.g. extraction of class diagrams

Type Reconstruction
- input: program without types
- output: program with types
Trade-offs

- Precision vs. efficiency
  - We chose efficiency for usage in a development browser
  - Use Heuristics as basis for the reconstruction
    - instead of full reconstruction

Heuristics

- Direct sends to instance variable
- Indirect sends to instance variables (getter methods)
- Direct assignment expressions
- Indirect assignment expressions (setter methods)
  - (Type snooping)
Implementations

- Using LiCoR (Library for Code Reasoning) in SOUL
  - on the parse tree
  - average: 500 milliseconds / instance variable
  - more elaborate and easier to extend
- Using partial evaluation on the byte code
  - average: 30 milliseconds / class

Demo
Conclusions & Future Work

- **Works**
  - *About 80% of correctness on built-in libraries*
  - Better on domain-specific code

- **Future Work**
  - Fix *About*
  - Will do this on (untyped) Java code and compare results
Challenges in Software Evolution

Tom Mens

http://w3.umh.ac.be/genlog
Software Engineering Lab
University of Mons-Hainaut
Belgium

The presented results are the outcome of the ChaSE 2005 workshop
- Financed by ESF and ERCIM
- April 2005, Bern, Switzerland

Scientific goals
- to discuss about and identify the main challenges in software evolution
- to address the above goal from different points of view
Classification of challenges

- Multidimensional classification
  - Time dimension
    - Short-term, medium-term, long-term research
  - Type of software evolution research
    - Managing software evolution
    - Understanding software evolution
    - Analysing software evolution
  - Interested stakeholder
    - Manager, end-user, developer, teacher, tool builder, software engineer, ...
  - Type of artifact under study
    - Formalism, tool, model, language, process, people, ...
  - Type of support provided
    - Same list as before...

Preserving and improving sw quality

- How can we provide tools and techniques that preserve or even improve the software quality, whatever its size, complexity, level of abstraction?

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>Controlling &amp; supporting evolution</td>
<td>developer, project manager, end-user</td>
<td>Software system</td>
<td>tools, techniques, formalisms, processes</td>
</tr>
</tbody>
</table>
Supporting model evolution

- Design and modeling tools provide little evolution support
- Evolution techniques needed at higher level of abstraction
  - A&D models, SW architectures, requirements, ...
- Model-driven engineering makes this challenge very relevant

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>controlling,</td>
<td>software engineer</td>
<td>models</td>
<td>tools, techniques, formalisms</td>
</tr>
<tr>
<td></td>
<td>supporting</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Supporting co-evolution

- We urgently need better techniques to achieve co-evolution
  - Synchronisation, consistency maintenance, inconsistency management, traceability, change propagation, ...
- Between different types of software artifacts or different representations
  - Programs and design models
  - Software and the organisation
  - Software and languages, tools, platforms

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>medium</td>
<td>controlling,</td>
<td>software engineer</td>
<td>any pair of related artifacts</td>
<td>tools, techniques, formalisms</td>
</tr>
<tr>
<td></td>
<td>supporting</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Formal support for evolution

- Some formal methods not amenable for an evolutionary setting
  - e.g., no incremental verification
- Formalisms for specific evolution activities needed
  - e.g., for refactoring

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>medium - long</td>
<td>all</td>
<td>researcher</td>
<td>formalisms</td>
<td>formalisms</td>
</tr>
</tbody>
</table>

Need for empirical research

- empirical and statistical studies needed to assess impact of process models, tools, languages, and people’s experience on software evolution

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>analysing</td>
<td>researcher</td>
<td>any evolving artifact</td>
<td>statistical models, empirical studies</td>
</tr>
</tbody>
</table>
Need for evolution benchmarks

• Commonly agreed representative benchmark or case studies to compare the developed formalisms, tools, techniques on relevant and typical problems
• Getting data from industrial setting is not easy, but there are many open-source, long-lived, industrial size projects

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>short-</td>
<td>understanding,</td>
<td>researcher</td>
<td>software</td>
<td>benchmarks,</td>
</tr>
<tr>
<td>medium</td>
<td>comparing</td>
<td></td>
<td>system</td>
<td>exemplars, cases</td>
</tr>
</tbody>
</table>

Bram Adams

Runtime evolution

• There is a need for proper support of runtime adaptations that allow systems to evolve while they are running, without needing to pause them or shut them down
  - Reflective techniques, metadata

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>short-</td>
<td>Controlling,</td>
<td>developer,</td>
<td>languages,</td>
<td>languages, execution</td>
</tr>
<tr>
<td>medium</td>
<td>supporting</td>
<td>end-user</td>
<td>execution</td>
<td>platforms, programs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>platforms</td>
<td></td>
</tr>
</tbody>
</table>

Challenge: Teaching software evolution

• How can we introduce the ideas and techniques of evolution into our educational system?
  - What do we want to teach to our students?
  - How can we teach this?
  - Where does it fit in the CS curriculum?

A common software evolution platform

• Proposed research solutions need to scale up to long-lived industrial-size software system
• Required tools are too complex to be built in isolation
• Need for a common platform, tool integration, exchange formats, standards and so on
• Candidates: MOOSE, Eclipse
Evolution as a language construct

- Change should be a first-class entity in programming or modelling languages
- Evolution support is easier in dynamically typed languages with reflective capabilities

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>short - medium</td>
<td>controlling, supporting</td>
<td>language designer, tool builder, researcher</td>
<td>languages</td>
<td>languages and programs</td>
</tr>
</tbody>
</table>

Supporting multi-language systems

- Many complex and large systems are built using 3 or more languages
- Evolution techniques should be parameterisable on or independent of the language

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>medium - long</td>
<td>controlling, supporting</td>
<td>tool builder</td>
<td>languages, software systems</td>
<td>tools, standards</td>
</tr>
</tbody>
</table>
Integrating change into development process

- Change must be integrated into conventional development process models
- Some, like agile development, already embrace change as essential
- Others, like the staged life-cycle model, have explicit support for evolution

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>medium</td>
<td>managing, controlling</td>
<td>manager, software engineer</td>
<td>software process model</td>
<td>software process model</td>
</tr>
</tbody>
</table>

Increasing manager’s awareness

- Managers have to realise the importance and inevitability of software evolution
- Teach them how to plan, organise and control projects to cope with change

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>motivating</td>
<td>manager, researcher</td>
<td>metaphors</td>
<td></td>
</tr>
</tbody>
</table>
Developing better versioning systems

- Many analyses of software evolution based on CVS or related tools
  - these weren’t built for that purpose and don’t store enough information
- New techniques and tools needed for recording the evolution of a system
- SCM is related

<table>
<thead>
<tr>
<th>Time</th>
<th>Research Type</th>
<th>Stakeholder</th>
<th>Artifact</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>Analysing</td>
<td>Tool Builder</td>
<td>Version Control Tools</td>
<td>Tools</td>
</tr>
</tbody>
</table>

Integrating and analysing evolutionary data

- Scattered information about system changes
  - bug reports, source code, documentation, configuration files, ...
- Very large data sets, especially for long-lived systems
- Need efficient and heterogeneous techniques
- Extensible meta-models, data mining, and bioinformatics may be relevant

<table>
<thead>
<tr>
<th>Time</th>
<th>Research Type</th>
<th>Stakeholder</th>
<th>Artifact</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium</td>
<td>Analysing</td>
<td>Researcher</td>
<td>All relevant info about sw system’s evolution</td>
<td>Techniques, Tools</td>
</tr>
</tbody>
</table>
A theory of software evolution

- Lehman has developed laws, but they need to be formalised and enriched
- May borrow ideas from evolutionary biology or linguistic evolution
- The what (noun) and how (verb) of evolution are still mostly disconnected
- How does the gathered data inform tools, techniques and formalisms?

<table>
<thead>
<tr>
<th>time</th>
<th>research type</th>
<th>stakeholder</th>
<th>artifact</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>understanding, analysing</td>
<td>researcher</td>
<td>everything</td>
<td>theories, formalisms, laws, ...</td>
</tr>
</tbody>
</table>

Software Evolution Case Studies

Filip Van Rysselberghe

Benevol 2005

Case study

“Aspect mining research results will have to be validated by means of a series of case studies.”

= illustrate the applicability on a concrete project
Examples

Benevol

ASML
a Perl Program
DocGen
Ant
LanSim
Tomcat

Icsm04

Mozilla
CodeCrawler
Linux
Apache
Gcc

Comparison

✍ A case per group/technique

• We can’t compare using a different basis!
• We can’t build on each other’s findings

✍ Are we really that different?

• Understanding, quality of architectures, aspect mining,..
• It’s all about Software Evolution!
⇒ Similar solutions?
⇒ Common problems?
LanSimulation

Which components are strongly coupled?

Representative

• Common evolution characteristics?
  – What’s typical/common?
  – Enough information about evolution?

• Characteristics of the case?
  – How do we measure?
  – Amount of info available?
  – When?
### Retrospective View

**Common**

- Life-cycle
- Constant change
- Feature requests
- Large/Coarse

**Case (Tomcat)**

- Build from previous product
- Constant change
- Implements specification
- Limited/Coarse
- Java
- Design docs/rationale
- Mailing Archives
Your characteristics

Is this typical?

- 450 LOC
- 19,000 LOC
- 90,000 LOC
- 170,000 LOC

- Perl
- Java
- C

- XP-methodology
- Tests integrated
- Ex-research. prototype

Literature characteristics

- System serves a user base
- User base has changing demands
- Generally grows
- Constant change (bugs/functional)
Case Selection

• How do we select our cases?
  – Based on measures? Which?
  – Based on quality of information?
• Where do we search?
• How did you act in the past?

Maybe like this?

Popular projects (DLs/rankings)
State (#releases/Warnings/Beta?)
Size Evolution (age/normal trend)
User impact (reporting/#MRs)
Changes (change logs)
Conclusion

• Many open questions!
  – Common Characteristics?
  – Case Selection?

**Discuss**

speak with others about it;
talk it over in detail;
have a discussion;
Introduction - Software Evolution

- More and better tool support needed for software evolution
  - At all levels of abstraction (e.g. programs and models)
  - For a variety of different activities
Introduction - Software Evolution

• Formalisms can be helpful for such evolution support
  - Description logics
    • For model inconsistency management
      - collaboration with R. Van Der Straeten, VUB
  - Graph transformation
    • For supporting software refactoring
    • Reasoning about preservation properties
      - collaboration with D. Janssens and S. Demeyer, UA
    • Analysing refactoring dependencies
      - collaboration with G. Taentzer and O. Runge, TU Berlin

Graph transformations

• GT theory theoretical results can help during analysis of model refactorings
  - type graph, negative application conditions, parallel and sequential (in)dependence, confluence and critical pair analysis
• GT tools allow us to perform concrete experiments
  - AGG (in collaboration with Berlin)
• Current focus
  - Analysing dependencies between class diagram refactorings
Analysing refactoring dependencies

• Concrete Scenario: Suggest refactoring opportunities
  - What are the alternatives of a selected refactoring?
  - Which other refactorings need to be applied first in order to make the selected refactoring applicable?
  - Which other refactorings are still applicable after applying the selected refactoring?

• Goal: Automate the detection of
  - mutual exclusion relationships between refactorings
  - sequential dependencies between refactorings

Analysing refactoring dependencies

• Example
Analysing refactoring dependencies

- Refactoring opportunities
  T1 Rename Method print in PrintServer to process
  T2 Rename Method save in FileServer to process
  T3 Create Superclass Server for PrintServer and FileServer
  T4 Pull Up Method accept from PrintServer and FileServer to Server

- Refactoring opportunities
  T5 Move Method accept from PrintServer to Packet
  T6 Move Method accept from FileServer to Packet
  T7 Encapsulate Variable receiver in Packet
  T8 Add Parameter p of type Packet to method print in PrintServer
  T9 Add Parameter p of type Packet to method save in FileServer
Analysing refactoring dependencies

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>×</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td>×</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td>×</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
</tr>
<tr>
<td>T4</td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
</tr>
<tr>
<td>T5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td>←</td>
<td>←</td>
<td>←</td>
<td>←</td>
</tr>
<tr>
<td>T6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td>←</td>
<td>←</td>
<td>←</td>
</tr>
<tr>
<td>T7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td>←</td>
<td>←</td>
</tr>
<tr>
<td>T8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td>←</td>
</tr>
<tr>
<td>T9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
</tr>
</tbody>
</table>

Applying graph transformation theory

- **Approach:** Use critical pair analysis in AGG
  - $T_1$ and $T_2$ form a critical pair if
    - they can both be applied to the same initial graph $G$ but
    - applying $T_1$ prohibits application of $T_2$ and/or vice versa
Applying graph transformation theory

Step 1: Express object-oriented metamodel as (attributed) type graph

Interludium

• Type graphs versus metamodels
Applying graph transformation theory

Step 2: Express refactorings as (typed attributed) graph transformations

Step 3: Detect critical pairs between refactoring transformations - Potential conflicts between refactorings
Applying graph transformation theory

Step 4: Fine-tune critical pairs in context of concrete input graph
Applying graph transformation theory

• Step 5: Perform sequential dependency analysis
  To identify dependencies between refactorings that are applicable

Conclusion

• Graph transformation theory is a suitable formalism for understanding software refactoring

<table>
<thead>
<tr>
<th>Graph Transformation</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>type graph, invariants</td>
<td>wf-constraints</td>
</tr>
<tr>
<td>negative application conditions</td>
<td>preconditions</td>
</tr>
<tr>
<td>parameterised graph production with NACs and context conditions mechanism</td>
<td>Refactoring transformation</td>
</tr>
<tr>
<td>Critical pair analysis</td>
<td>Detecting mutual exclusion</td>
</tr>
<tr>
<td>Confluence analysis</td>
<td>Detecting sequential dependencies</td>
</tr>
</tbody>
</table>
Refactoring Architectural Style
using KWIC as a case-study

Marc van Kempen
Michel Chaudron

Outline

Introduction / Motivation
Case Study: KWIC
Basics of CSP
Pipe & Filter → BlackBoard
Client-Server → BlackBoard
Correspondence to implementation
Conclusions
Introduction

What is Refactoring?

What is Architectural Style?
- Codification of common recurring patterns of software design
- Construction paradigms for (a set of) design dimensions
- A vocabulary of components, connectors and constraints on these can be combined (structure).
- Examples: Pipe and Filter, Client/Server, Blackboard

Motivation/Contributions

Motivation:
- Architecture-level refactoring
  - Useful for system migration, integration of large systems
  - Repair “mistakes” early in the development process
- Understanding architectural styles
  - Relation between styles (commonalities/differences)
- Research Separating Styles from Components
  - Exogeneous styles enhance component usability
  - Avoid architectural mismatch ⇒ ease evolution
**Approach**

Inspired by Darwin & CSP

- **Generic Model of Arch. Style X**
  - Instantiate
- **Generic Refactoring**
- **Generic Model of Arch. Style Y**
  - Instantiate

- **KWIC Model of Arch. Style X**
  - KWIC Refactoring
- **KWIC Model of Arch. Style Y**

---

**Connecting Connectors and Components**

What is a connector?
- Intermediates messages between components
- Shields component from interaction protocol

What is a component?
- Lots of definitions: "A unit of composition with clearly specified interfaces and explicit context dependencies only"
- Aim: components should be reusable; hence unchanged by refactoring

Use (adapted) Darwin notation to describe connections between connectors and components.
Keyword In Context (KWIC)

Based on a paper by Parnas (1972)

Garlan & Shaw (1994) use KWIC to illustrate the influence of Architectural Styles on Architectural Design

Good case to study refactoring between styles

KWIC (2)

What is KWIC?

Algorithm to generate alphabetically sorted permutations of an input sentence.

A permutation of a sentence is a sentence where the words are shifted one position to the right, the rightmost word is put at the beginning of the sentence. Repeat until the original sentence is reached.
KWIC(3)

Example input:
"Refactoring architectural styles"

Permutations:
"Refactoring architectural styles"
"Styles refactoring architectural"
"Architectural styles refactoring"

KWIC(4)

Sorted this becomes
"Architectural styles refactoring"
"Styles refactoring architectural"
"Refactoring architectural styles"

Which is also the expected output of the kwic process
CSP

What is CSP? Communicating Sequential Processes

Process algebra

Main operators:

Prefix: \( a \rightarrow P \)

Parallel composition: \( || \)

Communication actions: \( \text{out}!v \) and \( \text{in}?x \)

External choice: \( (a \rightarrow P) || (b \rightarrow Q) \)

Why use CSP to describe the processes?

Formalization

Tools exist to prove properties of model

Pipe and Filter

What is pipe and filter?
Pipe and Filter architecture

CSP definition of Pipe and Filter

Define

\[
\begin{align*}
\text{Filter} &= \text{in}\,x \rightarrow \text{outf}(x) \rightarrow \text{Filter} \\
\text{IN} &= \text{outf}\,x \rightarrow \text{IN} \\
\text{OUT} &= \text{in}\,x \rightarrow \text{OUT} \\
\text{Connector} &= C(\alpha) \\
C(\alpha) &= \text{in}\,x \rightarrow C(\alpha) \\
C(s) &= \text{in}\,x \rightarrow C(s^{\alpha}\langle x \neq \text{eof} \rangle) \\
C'(\alpha) &= \text{SKIP} \\
C'(s) &= \text{out}\text{head}(s) \rightarrow C'(\text{tail}(s))
\end{align*}
\]

Let

\[
\begin{align*}
\text{Filter} F1, F2 \\
\text{Connector} C1 \\
\text{IN}.\text{out} &\rightarrow F1.\text{in} \\
F1.\text{out} &\rightarrow C1.\text{in} \\
\text{etc.}
\end{align*}
\]

Then

\[
\text{PF} = \text{IN} \parallel F1 \parallel C1 \parallel F2 \parallel \text{OUT}
\]
Blackboard Architecture Style

CSP definition of Blackboard

IN, OUT, Filter remain the same!

Define
- Connector = in?x → ctrlx → outx → Connector
- BB(x) = in?x → BB(x)
- BB(s) = (in?x → BB(s^<x>)) [] (out!head(s) → BB(tail(s)))
- Controller = c1?x → Controller <| x ≠ eof | Controller1
- Controller1 = c2?x → c3?x → Controller1 <| x ≠ eof | Controller2
- Controller2 = c4?x → Controller2 <| x ≠ eof | SKIP

Let
- Filter F1, F2, F3
- Connector C1, C2, C3, C4
- Controller Ctrl
- BB BB1, BB2

Then
- Blackboard = IN || F1 || C1 || BB1 || C2 || F2 || C3 || BB 2 || C4 || OUT
KWIC in Pipe and Filter architecture style

Refactor P&F to BB
Refactor Client-Server to Blackboard

Client/Server refactored to Blackboard
In Java

Java implementation is forthcoming, watch website.

Contributions

Conclusions/Contributions

Distinguish control-flow, data-flow, reference-flow

It is possible to design 'style-agnostic components'

Styles are a means for glueing together components
Visualization of Software Evolution

Lucian Voinea
Alex Telea

- Benevol 2005 -
Eindhoven, Netherlands
27.05.2005

Visit us @ www.win.tue.nl/vis/

Challenge

Maintenance costs / Total costs > 90%

Bug discovery = 70 – 90% time

Code Analysis = 47 % time

Visit us @ www.win.tue.nl/vis/
Challenge

Initial development — Maintenance

Team A

Knowledge

Team B

Knowledge

Visit us @ www.win.tue.nl/vis/

Outline

Demo

Visit us @ www.win.tue.nl/vis/
Version layout – line based

Global line position

Discrete time (versions)

Local line position

Inserted lines

Version layout – interpolated

Global line position

Discrete time (versions)

Local line position

Inserted lines

Interpolated position

Empty space size decrease

Left bound empty space

Focus version

Right bound empty space

Visit us @ www.win.tue.nl/vis/
Version layout - interpolated

Visual improvements

- **Antialiasing**: Position based (to preserve structure)
- **Stable block detection**: Cushion based
Interaction

Zoom: Custom + predefined (fit to screen, fit to line)
Selection: Evolution interval selection

Navigation:

Filtering:

Visit us @ www.win.tue.nl/vis/
Tool presentation

- Control panel
- Evolution view
- Bi-level code view
- Metric

Visit us @ www.win.tue.nl/vis/

Demo

Visit us @ www.win.tue.nl/vis/
Conclusions

- Conveys process information embedded in the source code evolution records
- Offers version centric views on the system for relative assessment
- Appropriate mechanisms to navigate and browse the representation

Future work

- Visualize the evolution of more files in parallel
- Enrich the information about file structure
  - requires language specialization
Thank you for your attention!

www.win.tue.nl/~lvoinea/vcn.html
Bad smells and Refactorings

Let’s answer 4 major questions

Answering 4 questions

- Are there relations between bad smells?
- Are there relations between refactorings?
- Are there more relations between bad smells and refactorings?
- Do refactorings really take away bad smells?
Relations between bad smells?

- Model each bad smell as an IV
- Mine for relations
- Find relations between bad smells
- Check why intuitive relations do not hold
- Fine-tune Bad smells Model

Relations between refactorings?

- Model the refactoring prerequisites as IVs
- Mine for relations on a SW application
- Find relations between refactorings
Relations between bad smells and refactorings?

- Model each bad smell as an IV
- Model the different refac. prerequisites as IVs
- Mine for relations on a SW application
- Answers the question whether we really can apply all refactorings (Fowler suggested) for solving a bad smell

Do refactorings really take away bad smells?

- Model each bad smell as an IV
- Find all bad smells in all CVS versions of a SW application and establish a "bad smell occurrence curve"
- Check the CVS comments to see whether refactorings were carried out on which points
Understanding Change

Where do we look at?
Filip Van Rysselberghe

Benevol 2005

Background

*which change operations (and sequences) are applied?*

- Many changes
  - Which ones are relevant?
- Relevant changes
  - Changes to remove design problems
  - Changes that influence the evolution
  - Refactorings
Approaches

- **Problem-up**
  - Focus a change when it removes a problem
- **Evolution-up**
  - Focus a change when its evolution changes
- **Rationale-up**
  - Focus a change when the developer tags it

Experimented with other code smells as well.
Problem-up experiences

- Manual validation
  - Time intensive
  - Need for a high level representation
  → Can graphs help us out?
- Change rationale
  - Why is not always as obvious
  - Link with additional info! Developers!

Evolution-up
Evolution-up experiences

- Can we put it in a number?
  - Change rate?
  - Patterns?
- Suitable to study the effect of ...
  - A change?
  - A code smell/problem?
- Links to the rationale
  - Learns us more about the why!
- Still have to study actual changes
  - High-level change representation!

Rationale-up

<table>
<thead>
<tr>
<th>Quality</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Easier</td>
<td>1. Move</td>
</tr>
<tr>
<td>2. Move</td>
<td>2. Improve</td>
</tr>
<tr>
<td>3. Refactor</td>
<td>3. Refactor</td>
</tr>
<tr>
<td>4. Clean</td>
<td></td>
</tr>
</tbody>
</table>

Tomcat:
- Refactor dependencies
- Move & Update

<table>
<thead>
<tr>
<th>project</th>
<th>MRs</th>
<th>refactorMRs</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>8127</td>
<td>210</td>
<td>0.5%</td>
</tr>
<tr>
<td>Gatm</td>
<td>906</td>
<td>4</td>
<td>0.04%</td>
</tr>
<tr>
<td>Jakarta-Tomcat</td>
<td>4098</td>
<td>60</td>
<td>1.42%</td>
</tr>
<tr>
<td>Jboss</td>
<td>7437</td>
<td>63</td>
<td>0.84%</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>20043</td>
<td>21</td>
<td>0.10%</td>
</tr>
</tbody>
</table>
Rationale-up Experiences

- Differences in CVSquality
  - Quick assessment of messages
- Branching is a problem
- High level change view!
- Rationale!

Conclusion