Mining Software Repositories

Mircea Lungu
Roadmap

> **Introduction**
> **Recovering entity evolution**
>   — Origin analysis
>   — Refactoring detection
> **Mining the history for relationships**
>   — Logical coupling
>   — Change propagation
> **Mining a history for rules**
>   — Common error patterns
>   — Associating artefacts with tasks
> **And more...**
Software is Data...

> Data that you analyze
> Data that you measure
> Data that evolves and can be mined
> ...
> Data that is big data
> Data that you visualize
A History of History Analysis in Software

> 20 years of **software engineering** before people start doing research in **analyzing software repositories**
References

Main Materials

— An Integrated Approach for Studying Architectural Evolution, Tu & Godfrey, ’02
— Automated Detection of Refactorings in Evolving Components, Dig et al., ’06
— Detection of Logical Coupling Based on Product Release History, Gall et al., ’98
— Predicting Change Propagation in Software Systems, Hassan & Holt, ’04
— DynaMine: finding common error patterns by mining software revision histories, Livshits & Zimmerman, ’05
Roadmap

> Introduction

> Recovering entity evolution
   — Origin analysis
   — Refactoring detection

> Mining the history for relationships
   — Logical coupling
   — Change propagation

> Mining a history for rules
   — Common error patterns
   — Associating artefacts with tasks

> And more...
A Quick Softwarenaut Demo...

> ... showing that new classes appear all the time during the evolution of the system

> But are they really “new”?
Origin Analysis

> Tu & Godfrey ’02
> Works at the function level
> Combines
>   - Bertillonage Analysis
>     - Assumes that Complexity Metrics do not change much
>   - Dependency Analysis
>     - Assumes that relationships do not change much
>   ...
Refactoring Detection

> Dig et al. ’06 detect refactorings of Packages, Classes, Methods
> Combination of **syntactic** and **semantic** analysis
> Shingles Algorithm

How to detect refactorings in object-oriented systems?
The Approach of Dig et al.: The Shingles Algorithm

> Input
  — sequence of tokens representing method body without signature

> Output
  — Multi-sets of integeres
  — Similar inputs generate similar outputs

> Algorithm
  — $W$: window size
  — $S$: maximum set size
  — Compute hashes while sliding the window
  — Sort shingles and keep the first $S$

The original Shingles paper:
Broder, On the resemblance and containment of documents
The Approach of Dig et al.: Shingles Algorithm (Example with W=2 and S=10)
The Approach of Dig et al.: Semantic Analysis

Seven Detection Strategies — applied in order — based on dependencies between artifacts
- method calls
- subclassing
- fields
- arguments
- parameters

1. RenamePackage (RP)
2. RenameClass (RC)
3. RenameMethod (RM)
4. PullUpMethod (PUM)
5. PushDownMethod (PDM)
6. MoveMethod (MM)
7. ChangeMethodSignature (CMS)
The Approach of Dig et al.: Results

> More than 85% Precision and Recall on
  — Eclipse
  — Struts
  — HotDraw

> What’s next? CatchUp!
Roadmap

> Introduction
> Recovering entity evolution
  — Origin analysis
  — Refactoring detection
> Mining the history for relationships
  — Logical coupling
  — Change propagation
> Mining a history for rules
  — Common error patterns
  — Associating artefacts with tasks
> And more...
Logical Coupling

> Gall et al. ‘98
> Based on an industrial case study
  — Subsystems
  — Modules
  — Programs
> Two steps
  1. Change Sequence Analysis
  2. Change Report Analysis

How to detect dependencies based on history?
Why history based?

> Structural / Data Flow Analysis

  – Disadvantages:

    – can not capture all the situations (i.e. writing to a file, reading from a file)

    – does not work with documents that are not source code
Change Sequence Analysis

- Detects when two subsystems change together
- Logical coupling is stronger if the subsequence is larger
Change Report Analysis

> There are two types of changes that are documented
  — Feature additions
  — Bug Requests

> The coupling between subsystems must be verified
Logical Coupling Summary

> Advantages
  — Does not require the code to compile
  — Can work with any types of documents

> Simplification
  — Versioning systems in the real world are a mess (CVS)
Change Propagation

> Hassan & Holt '04
> Compare heuristics
  — Developer (DEV)
  — Historical co-change (HIS)
  — Structural: Call/Use/Define (CUD)
  — Code layout (FIL)

What other entities have to change when a given one changes?
Evaluating the heuristics based on history with precision and recall

> Precision
> Recall
> Compute for every relevant change set and average

<table>
<thead>
<tr>
<th>Application</th>
<th>DEV</th>
<th>HIS</th>
<th>CUD</th>
<th>FIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall</td>
<td>Precision</td>
<td>Recall</td>
<td>Precision</td>
</tr>
<tr>
<td>NetBSD</td>
<td>0.74</td>
<td>0.01</td>
<td>0.87</td>
<td>0.06</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>0.68</td>
<td>0.02</td>
<td>0.87</td>
<td>0.06</td>
</tr>
<tr>
<td>OpenBSD</td>
<td>0.71</td>
<td>0.02</td>
<td>0.82</td>
<td>0.08</td>
</tr>
<tr>
<td>Postgres</td>
<td>0.78</td>
<td>0.01</td>
<td>0.86</td>
<td>0.05</td>
</tr>
<tr>
<td>GCC</td>
<td>0.79</td>
<td>0.01</td>
<td>0.94</td>
<td>0.03</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.74</td>
<td>0.01</td>
<td>0.87</td>
<td>0.06</td>
</tr>
</tbody>
</table>
Hybrid Technique

Entities that changed together at least A% of the time (prune HIS) OR Entities in the same file that changed together at least A%-B% of the time (FIL)
Change Propagation Discussion

- Heuristics
  - Only work with one element in the prediction set
  - Are symmetric
- File-level is a limitation
Roadmap

> Introduction
> Recovering entity evolution
  — Origin analysis
  — Refactoring detection
> Mining the history for relationships
  — Logical coupling
  — Change propagation
> **Mining a history for rules**
  — Common error patterns
  — Associating artefacts with tasks
> And more...
Common Error Patterns

- Livshits & Zimmermann ’05
- Data mining reveals frequent patterns
  - Matching Method Pairs
  - State Machines

How to detect bugs in apps that use APIs about which you do not have knowledge?
Principles

1. API specific errors
2. Co-addition is a pattern
3. Small commits are fixes
When to look for pattern violations?

> Runtime
  + Scalability
  + Simplicity (no interprocedural analysis)
  + Counting occurrences
  + Zero False Positives
  – Coverage
Dynamine: The Approach

- Human Input is required
- Mines from the history
- Validates at runtime
Mining for Likely Patterns: The Apriori Algorithm

> Concepts
  — Usage Pattern
  — Transaction
  — Support Count

> Input
  — Minimum Support

> Output
  — Frequent Patterns

> Implementation
  — Iterative
  — Exponential
Pattern Filtering

> Consider a subset of the methods
  — ignore initial revisions
  — ignore common calls
> Consider small patterns only
  — group calls by access path
Pattern Ranking & Classification

> Lexicographically on support count

> Corrective ranking
   — assumes on one-line changes are bug-fixes
   — used as first lexicographic category improves bug finding

> Classification
   — Usage
   — Error
   — Unlikely
# Results

## Eclipse

<table>
<thead>
<tr>
<th>Method pair (a, b)</th>
<th>Confidence</th>
<th>Support</th>
<th>Dynamic</th>
<th>Static</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Eclipse</strong> (16 pairs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NewRgn, DisposeRgn</td>
<td>0.76</td>
<td>0.92</td>
<td>0.82</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>kEventControlActivate, kEventControlDeactivate</td>
<td>0.69</td>
<td>0.83</td>
<td>0.83</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>addDebugEventListener, removeDebugEventListener</td>
<td>0.61</td>
<td>0.85</td>
<td>0.72</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>beginTask, done</td>
<td>0.60</td>
<td>0.74</td>
<td>0.81</td>
<td>493</td>
<td></td>
</tr>
<tr>
<td>beginRule, endRule</td>
<td>0.60</td>
<td>0.80</td>
<td>0.74</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>suspend, resume</td>
<td>0.60</td>
<td>0.83</td>
<td>0.71</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>NewPtr, DisposePtr</td>
<td>0.57</td>
<td>0.82</td>
<td>0.70</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>addListener, removeListener</td>
<td>0.57</td>
<td>0.68</td>
<td>0.83</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>register, deregister</td>
<td>0.54</td>
<td>0.69</td>
<td>0.78</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>malloc, free</td>
<td>0.47</td>
<td>0.68</td>
<td>0.68</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>addElementChangedListener, removeElementChangedListener</td>
<td>0.42</td>
<td>0.73</td>
<td>0.57</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>addResourceChangeListener, removeResourceChangeListener</td>
<td>0.41</td>
<td>0.90</td>
<td>0.46</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>addPropertyChangeListener, removePropertyChangeListener</td>
<td>0.40</td>
<td>0.54</td>
<td>0.73</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>start, stop</td>
<td>0.39</td>
<td>0.59</td>
<td>0.65</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>addDocumentListener, removeDocumentListener</td>
<td>0.36</td>
<td>0.64</td>
<td>0.56</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>addSyncSetChangeListener, removeSyncSetChangeListener</td>
<td>0.34</td>
<td>0.62</td>
<td>0.56</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

## jEdit

<table>
<thead>
<tr>
<th>Method pair (a, b)</th>
<th>Confidence</th>
<th>Support</th>
<th>Dynamic</th>
<th>Static</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>jEdit</strong> (8 pairs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addNotify, removeNotify</td>
<td>0.60</td>
<td>0.77</td>
<td>0.77</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>setBackground, setForeground</td>
<td>0.57</td>
<td>0.67</td>
<td>0.86</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>contentRemoved, contentInserted</td>
<td>0.51</td>
<td>0.71</td>
<td>0.71</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>setInitialDelay, start</td>
<td>0.40</td>
<td>0.80</td>
<td>0.50</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>registerErrorSource, unregisterErrorSource</td>
<td>0.28</td>
<td>0.45</td>
<td>0.62</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>start, stop</td>
<td>0.20</td>
<td>0.39</td>
<td>0.52</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>addToolBar, removeToolBar</td>
<td>0.18</td>
<td>0.60</td>
<td>0.30</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>init, save</td>
<td>0.09</td>
<td>0.40</td>
<td>0.24</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

Subtotals for the corrective ranking scheme: 5,546 2,051 241 222 3 U, 8 E
Associating Artefacts with Tasks

> Kersten & Murphy ‘05
> Mylar/Mylin
> Task-Focused Interface
> Degree of Interest ranking

How to filter the large amount of information available in the IDE?
Roadmap

> Introduction

> Recovering entity evolution
  — Origin analysis
  — Refactoring detection

> Mining the history for relationships
  — Logical coupling
  — Change propagation

> Mining a history for rules
  — Common error patterns
  — Associating artefacts with tasks

> And more...
Further Directions (Kagdi et al. ’07)

> Basic evolution principles
  — Refactorings breaking clients (Dig & Johnson ’05)
  — Understanding the rhetoric of small changes (Puru & Perry ’05)

> Change-Based repositories
  — Replay (Hattori et. al ’11)

> Bug prediction
  — Extensive comparison of approaches (D’Ambros et al. ’10)

> Risk Prediction
  — The Code Orb, (Lopez ’11)
Benefits of Historical Analysis

> Predict various aspects of the system based on the past
  — Temporal locality
  — Co-change patterns

> Increase the amount of available information
> Allows empirical validation of hypotheses
Enablers of Historical Analysis

> Versioning systems
> Increased amounts of historical data
> Availability of different types of data
  — developer interaction
  — bug/issue tracking
> Modern IDE’s
  — plugin philosophy
    – collecting data
    – playground for features
Roadmap

> Introduction
> Recovering entity evolution
  — Origin analysis
  — Refactoring detection
> Mining the history for relationships
  — Logical coupling
  — Change propagation
> Mining a history for rules
  — Common error patterns
  — Associating artefacts with tasks
> And more...
What you should know!

> What is origin analysis
> What is logical coupling
> How does the Apriori algorithm function
> What are shingles and how do they work
Can you answer these questions?

> How does origin analysis work in the approach of Tu & Godfrey?
> Can you compare the Bertillonage and the Shingles approaches?
> Why does the Dynamine tool require dynamic analysis?
> What heuristics would you use to predict classes that change together and why?
> Can you discuss some of the advantages and some of the disadvantages of the shingles technique?
Further Reading

- Mylar, a Degree of Interest model for IDE's, Kersten & Murphy ’05
- The Role of Refactorings in API Evolution, Dig & Johnson, ’05
- The code orb: supporting contextualized coding via at-a-glance views, Lopez ’11
- Modeling History to Understand Software Evolution, Girba, ’05
- An extensive comparison of bug prediction approaches, D’Ambros et al., ’10
- Software Evolution Comprehension: Replay to Rescue, Hattori et al., ’11
- A survey and taxonomy of approaches for mining software repositories in the context of software evolution, Kagdi et al. ’07
Attribution-ShareAlike 2.5

You are free:
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/2.5/