# Effective Mining of Software Repositories

Marco D'Ambros
REVEAL group
University of Lugano
Switzerland

Romain Robbes
PLEIAD @ DCC
University of Chile
Chile

# Objectives of the tutorial

- Provide a high-level view of the MSR area
- Present a number of MSR techniques
- Point at limitations and future research directions
- Provide an extensive reference set for further readings

# Topics covered

- The origins of MSR
- How to pre-process MSR data
- A selection of MSR approaches
- Limitations of MSR
- Solutions to these limitations

- Two versions of the tutorial
  - "Live" version: fits in 3 hours (hopefully :-) )
  - Print version, with more content (400+ slides)

# Topics covered

- The origins of MSR
- How to pre-process MSR data
- A selection of MSR approaches
- Limitations of MSR
- Solutions to these limitations

# Introduction

# What is MSR?

# MSR by example

The version editor, ca. 1988

```
String FindSource(String base, String dir) {
    DIR * dirp = opendir(dir);
    String result;     // The filename, if found
    for (int i = 0; i < NS; ++i) {        // Loop over suffix list
        String tmp = base + suffix[i];    // Target name to find
        for (dirent *de = readdir(dirp); de != NULL; de = readdir(dirp))
           if (tmp == de->d_name) {       // We found it, stop looking
               result = tmp;
               break;
               return tmp;
           }
        rewinddir(dirp);
    }
    closedir(dirp);
    return result; // Return the found name (may be null)
    return "";     // No match was found
}

Deleted by MR 595 by vz,97/11/15,approved [Stop source search at 1st match]
MR 467 by dla,97/09/21,integrated [Find source using list of suffixes]
"findsource.c", line 15 of 23
```

ve view (recent additions bold, deletions underlined)

# This looks boring, until ...

Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor *

David L. Atkins,* Thomas Ball,× Todd L. Graves+ and Audris Mockus°

* University of Oregon
× Microsoft Research
+ Los Alamos National Laboratory
° Avaya Labs Research

October 9, 2001

**ABSTRACT**

Software tools can improve the quality and maintainability of software, but are expensive to acquire, deploy and maintain, especially in large organizations. We explore how to quantify the effects of a software tool once it has been deployed in a development environment. We present an effort-analysis method that derives tool usage statistics and developer actions from a project's change history (version control system) and uses a novel effort estimation algorithm to quantify the effort savings attributable to tool usage.

We apply this method to assess the impact of a software tool called VE, a version-sensitive editor used in Bell Labs. VE aids software developers in coping with the rampant use of certain preprocessor directives (similar to #if/#endif in C source files). Our analysis found that developers were approximately 40% more productive when using VE than when using standard text editors.

**Keywords**

software tools, version control system, effort analysis

## 1 Introduction

While software tools have the potential to improve the quality and maintainability of software, acquiring, deploying and maintaining a tool in a large organization can be an expensive proposition. We explore how to quantify the effects of a software tool in an ongoing large-scale software project. We describe a case study of the impact of a version-sensitive text editor called VE. We assess the impact using a method that relates tool usage statistics with effort estimates based on analysis of the change history of a software project. The value in performing such an impact analysis is to create data from which subsequent decisions about the tool use can be made more effectively (e.g., to keep a tool, to deploy it more widely, to reward its use, to publish results that would influence other potential adopters, etc.)

Our work is based on two observations. The first observation is that a major effect of a software tool, be it a documentation tool, source code editor, code browser, slicer, debugger, or memory-leak detector, is to help a developer determine how to modify a software entity or directly to aid the developer in making modifications. The second observation is that the change history of a software entity (i.e., the version control data about the modifications to the entity) can be used to estimate the amount of effort a developer expended on a particular modification or set of modifications, as well as measures of the overall time (interval) taken to develop a software feature. To obtain accurate estimates of tool effects it is often important to have effort estimates at a fine grained change level, however, it is unreasonable to expect that developers could always accurately and efficiently report effort for individual changes they commit to a version control system[1].

These observations lead to a simple process for assessing the impact of a software tool:

1. Record the tools a developer uses in the course of software development and the software entities to which they were applied.
2. Relate the monitoring information recorded in step 1 to the modifications to software entities that are recorded by the version control system.
3. Using the data from the previous two steps and the change effort estimation algorithm described in Section 4, analyze "similar" developers and modifications[2]

*This work was done while all four authors were members of the Software Production Research Department in Lucent Technologies' Bell Laboratories.

[1] In Section 4 we introduce an algorithm that estimates effort for individual changes from information available in a version control system.

[2] Section 5 qualifies and quantifies the notions of "similar" developers and modifications.

Retrospective evaluation on 10 years of usage at Bell Labs leads to estimated savings of **1,400 person/years**.

# Wait a minute, how did they compute that?

> “[...] a noteworthy aspect of VE is that it leaves a signature on all of the #version control lines that it generates. This signature consists of trailing white space (a combination of space and tab characters) that uniquely distinguishes the control line from any control line generated for any other change.”

# So, what is MSR?

Mining software repositories analyzes the large quantities of data available in software repositories to:

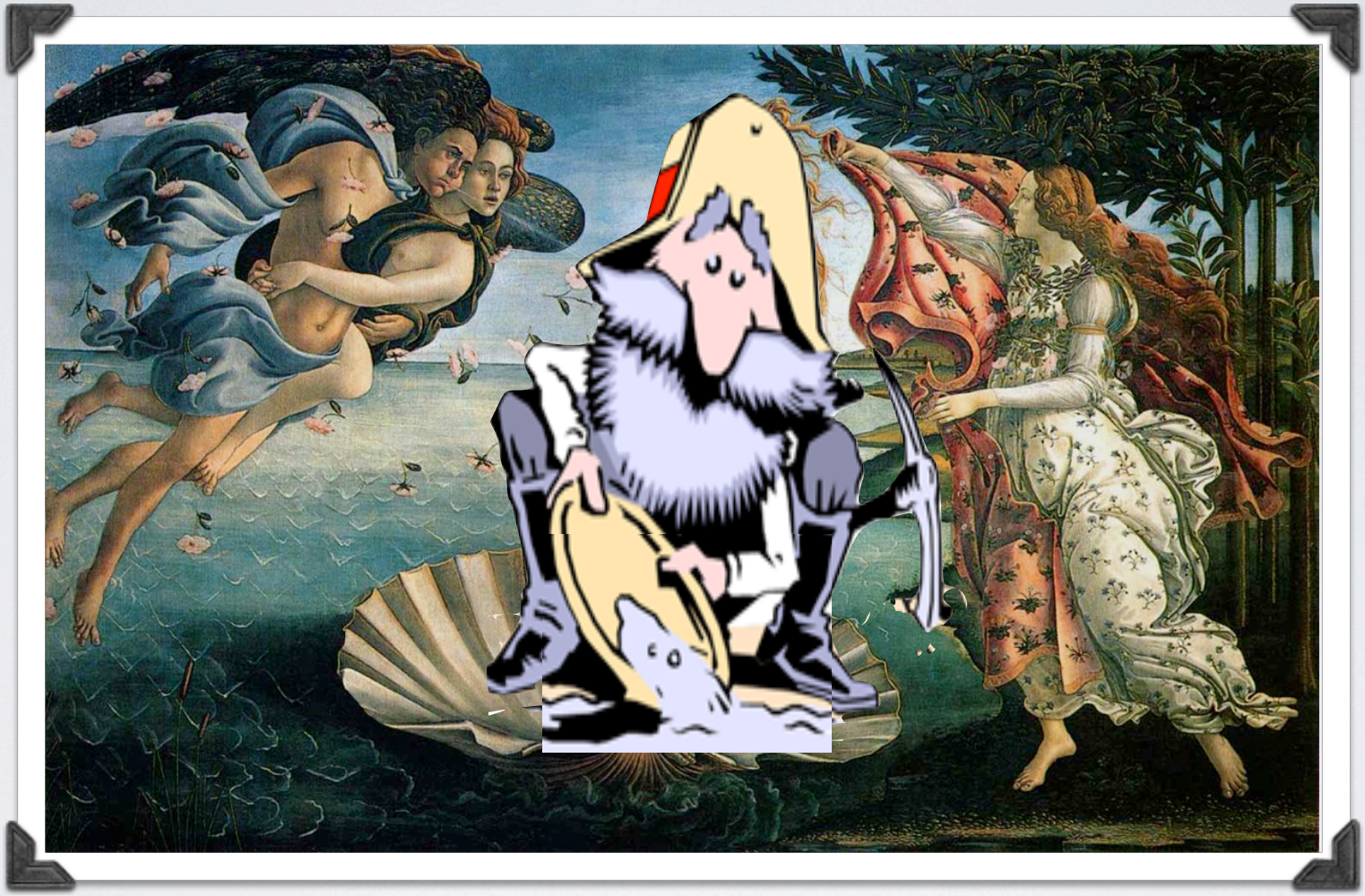
1) uncover or confirm facts about software systems and their evolution;
2) assist developers, managers, testers, etc, working on these systems;
3) perform predictions about the future of software systems.

# Outline

- Introduction to MSR
  - History of MSR
  - Common software repositories
  - Data preprocessing
- The state of the art in MSR
- Limitations of MSR
- Conclusions

# MSR: a bit of history

# The birth of MSR, ca. ~~1486~~ 1986



# MSR Timeline

OS/360 studies

69 · 75 76 · 80 · 82 · 86 · 88 · 92 · 97 98 99 00 · 03 04 05 · 08 ▶

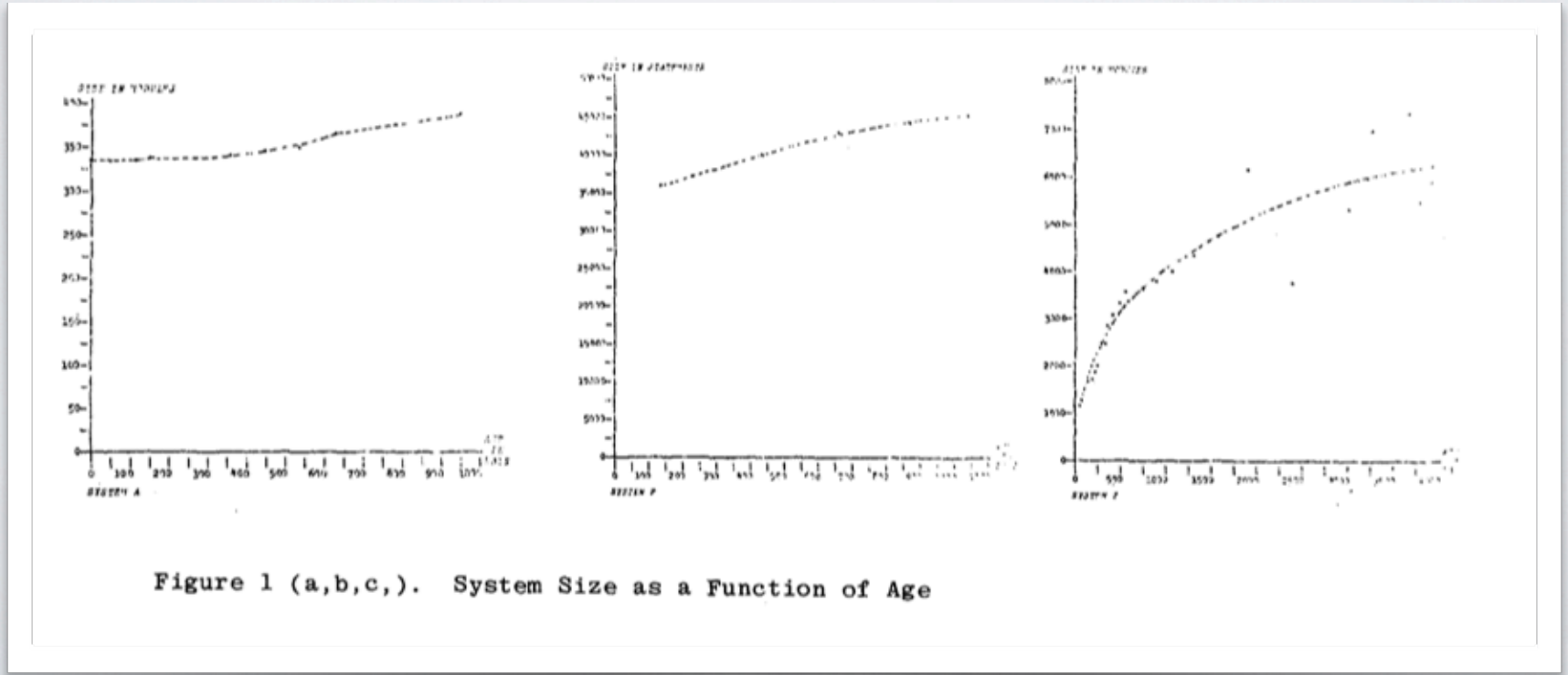
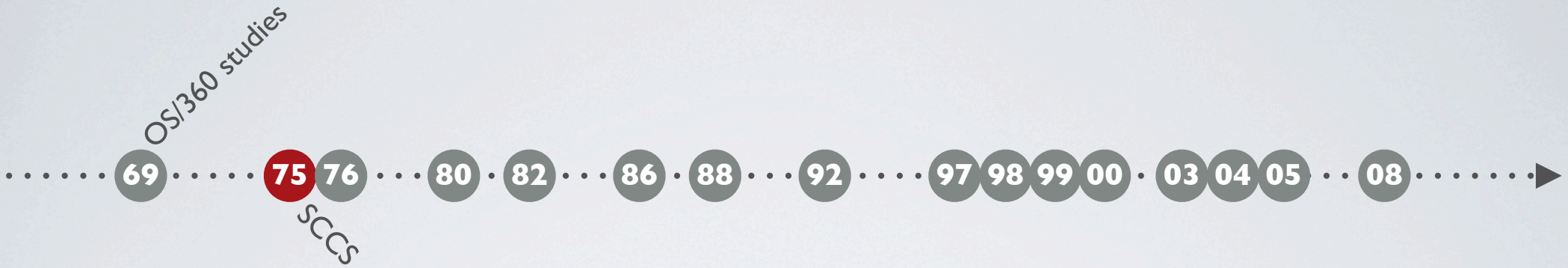


Figure 1 (a,b,c,). System Size as a Function of Age

# MSR Timeline






## The Source Code Control System

MARC J. ROCHKIND

*Abstract*—**The Source Code Control System (SCCS) is a software tool designed to help programming projects control changes to source code. It provides facilities for storing, updating, and retrieving all versions of modules, for controlling updating privileges, for identifying load modules by version number, and for recording who made each software change, when and where it was made, and why. This paper discusses the SCCS approach to source code control, shows**

*Identification:* The system automatically stamps load modules with information such as version number, date, time, etc. The source code that was used to make the load module may later be retrieved from this information alone.

*Documentation:* The system automatically records who

# MSR Timeline


THE DIMENSIONS OF MAINTENANCE

E. Burton Swanson
Graduate School of Management
University of California, Los Angeles
Los Angeles, California 90024

PROGRAM EVOLUTION AND ITS IMPACT ON

SOFTWARE ENGINEERING

M. M. Lehman

and

F. N. Parr

Department of Computing and Control
Imperial College of Science and Technology, London

*Keywo*

*Abstr*

ed by

# MSR Timeline




## Programs, Life Cycles, and Laws of Software Evolution

MEIR M. LEHMAN, SENIOR MEMBER, IEEE

*Abstract*—By classifying programs according to their relationship to the environment in which they are executed, the paper identifies the sources of evolutionary pressure on computer applications and programs and shows why this results in a process of never ending maintenance activity. The resultant life cycle processes are then briefly discussed. The paper then introduces laws of Program Evolution that have been formulated following quantitative studies of the evolution of a number of different systems. Finally an example is provided of the application of Evolution Dynamics models to program release planning.

### I. Background

#### A. The Nature of the Problem

THE TOTAL U.S. expenditure on programming in 1977 is

serious problems in the provision and upkeep of satisfactory programs. It also yielded new insights. Programming as then practiced required the breakdown of the problem to be solved into steps far more detailed than those in terms of which people thought about it and its solution. The manual generation of programs at this low level was tedious and error prone for those whose primary concern was the result; for whom programming was a means to an end and not an end in itself. This could not be the basis for widespread computer application.

Thus there was born the concept of *high-level*, problem-oriented, *languages* created to simplify the development of computer applications. These languages did not just raise the

# MSR Timeline


**RCS—A System for Version Control**

*Walter F. Tichy*

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

*ABSTRACT*

An important problem in program development and maintenance is version control, i.e., the task of keeping a software system consisting of many versions and configurations

# MSR Timeline


# MSR Timeline


```
String FindSource(String base, String dir) {
    DIR * dirp = opendir(dir);
    String result;    // The filename, if found
    for (int i = 0; i < NS; ++i) {        // Loop over suffix list
        String tmp = base + suffix[i];    // Target name to find
        for (dirent *de = readdir(dirp); de != NULL; de = readdir(dirp))
           if (tmp == de->d_name) {       // We found it, stop looking
               result = tmp;
               break;
               return tmp;
           }
        rewinddir(dirp);
    }
    closedir(dirp);
    return result; // Return the found name (may be null)
    return "";     // No match was found
}
```

# MSR Timeline


# MSR Timeline


If Your Version Control System Could Talk ...

Thomas Ball
Bell Laboratories
Lucent Technologies
tball@research.bell-labs.com

Jung-Min Kim Adam A. Porter
Dept. of Computer Sciences
University of Maryland
aporter@cs.umd.edu

Harvey P. Siy
Bell Laboratories
Lucent Technologies
hpsiy@research.bell-labs.com

**Abstract**

Version control systems (VCSs) are used to store and reconstruct past versions of program source code. As a by-product they also capture a great deal of contextual information about each change. We will illustrate some ways to use this information to better understand a program's development history.

## 1 Introduction

There are many software-based metrics that one may

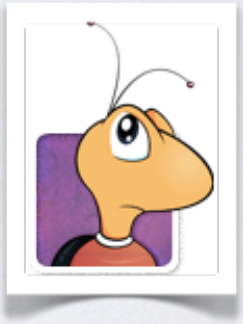
**Understanding and Predicting the Process of Software Maintenance Releases**

Victor Basili, Lionel Briand, Steven Condon,
Yong-Mi Kim, Walcélio L. Melo and Jon D. Valett†

**Abstract**

boundaries of resource constraints (such as, budget,

# MSR Timeline

## Inferring Change Effort from Configuration Management Databases

Todd L. Graves* and Audris Mockus+
*National Institute of Statistical Sciences and *+Bell Laboratories

**ABSTRACT**
In this paper we describe a methodology and algorithm for historical analysis of the effort necessary for developers to make changes to software. The algorithm identifies factors which have historically increased the difficulty of changes. This methodology has implications for research into cost drivers. As an example of a research finding, we find that a system under study was "decaying" in that changes grew more difficult to implement at a rate of 20% per year. We also quantify the difference in costs between changes that fix faults and additions of new functionality: fixes require 80% more effort after accounting for size. Since our methodology adds no overhead to the development process, we also envision it being used as a project management tool: for example, developers can identify code modules which have grown more difficult to change than previously, and can match changes to developers with appropriate exper-

in Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98) 

## Detection of Logical Coupling Based on Product Release History

Harald Gall, Karin Hajek, and Mehdi Jazayeri

*Technical University of Vienna, Distributed Systems Group*
*Argentinierstrasse 8/184-1, A-1040 Wien, Austria, Europe*
*{gall,hajek,jazayeri}@infosys.tuwien.ac.at*

# MSR Timeline




## Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor

David L. Atkins, Thomas Ball, Todd L. Graves, and Audris Mockus, *Member*, *IEEE*

**Abstract**—Software tools can improve the quality and maintainability of softw
especially in large organizations. We explore how to quantify the effects of a s
environment. We present an effort-analysis method that derives tool usage
history (version control system) and uses a novel effort estimation algorithm t
apply this method to assess the impact of a software tool called VE, a versi
developers in coping with the rampant use of certain preprocessor directives
that developers were approximately 40 percent more productive when using

**Index Terms**—Software tools, version control system, effort analysis.

### 1 Introduction

While software tools have the potential to improve the quality and maintainability of software, acquiring, deploying, and maintaining a tool in a large organization develop
modifi
(interv

**Yesterday, my Program Worked. Today, it Does Not. Why?**

Andreas Zeller

# MSR Timeline



## A Case Study of Open Source Software Development: The Apache Server

**Audris Mockus**
Bell Labs, 263 Shuman Blvd.
Naperville, IL 60566 USA
+1 630 713 4070
audris@research.bell-labs.com

**Roy T. Fielding**
Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
fielding@ics.uci.edu

**James Herbsleb**
Bell Labs, 263 Shuman Blvd.
Naperville, IL 60566 USA
+1 630 713 1869
herbsleb@research.bell-labs.com

**ABSTRACT**
According to its proponents, open source style software development has the capacity to compete successfully, and perhaps in many cases displace, traditional commercial development methods. In order to begin investigating such claims, we examine the development process of a major open source application, the Apache web server. By using email archives of source code change history and problem reports we quantify aspects of developer participation, core team size, code ownership, productivity, defect density, and problem resolution interval for this OSS project. This analysis reveals a unique process, which performs well on

that ensure that the source code for an OSS development will be generally available. Open source developments typically have a central person or body that selects some subset of the developed code for the "official" releases and makes them widely available for distribution.

These basic arrangements to ensure freely available source code have led to a development process that is radically different, according to OSS proponents, from the usual, industrial style of development. The main differences usually mentioned are

- OSS systems are built by potentially large numbers (i.e. hundreds or even thousands) of volunteers

# MSR Timeline


## Populating a Release History Database from Version Control and Bug Tracking Systems*

Michael Fischer, Martin Pinzger, and Harald Gall
Distributed Systems Group, Vienna University of Technology
{fischer,pinzger,gall}@infosys.tuwien.ac.at

### Abstract

*Version control and bug tracking systems contain large amounts of historical information that can give deep insight into the evolution of a software project. Unfortunately, these systems provide only insufficient support for a de-*

anticipating future evolution of software projects. Unfortunately, current version and bug report systems provide no or only insufficient support for the combination of both data sources and, hence, lack capabilities in software evolution analysis. Moreover, the formats of and access to version and bug report data vary across version control and bug re-

# MSR Timeline


# MSR Timeline



# MSR Timeline


# Mining **Which** Software Repository?

# Which software repository?

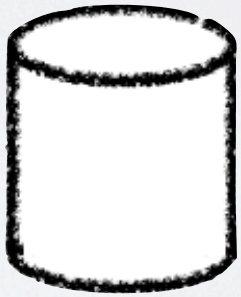
changes

bugs

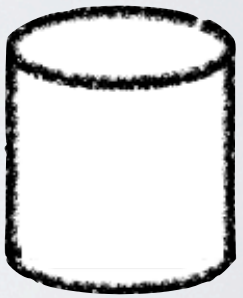
emails

models

execution traces

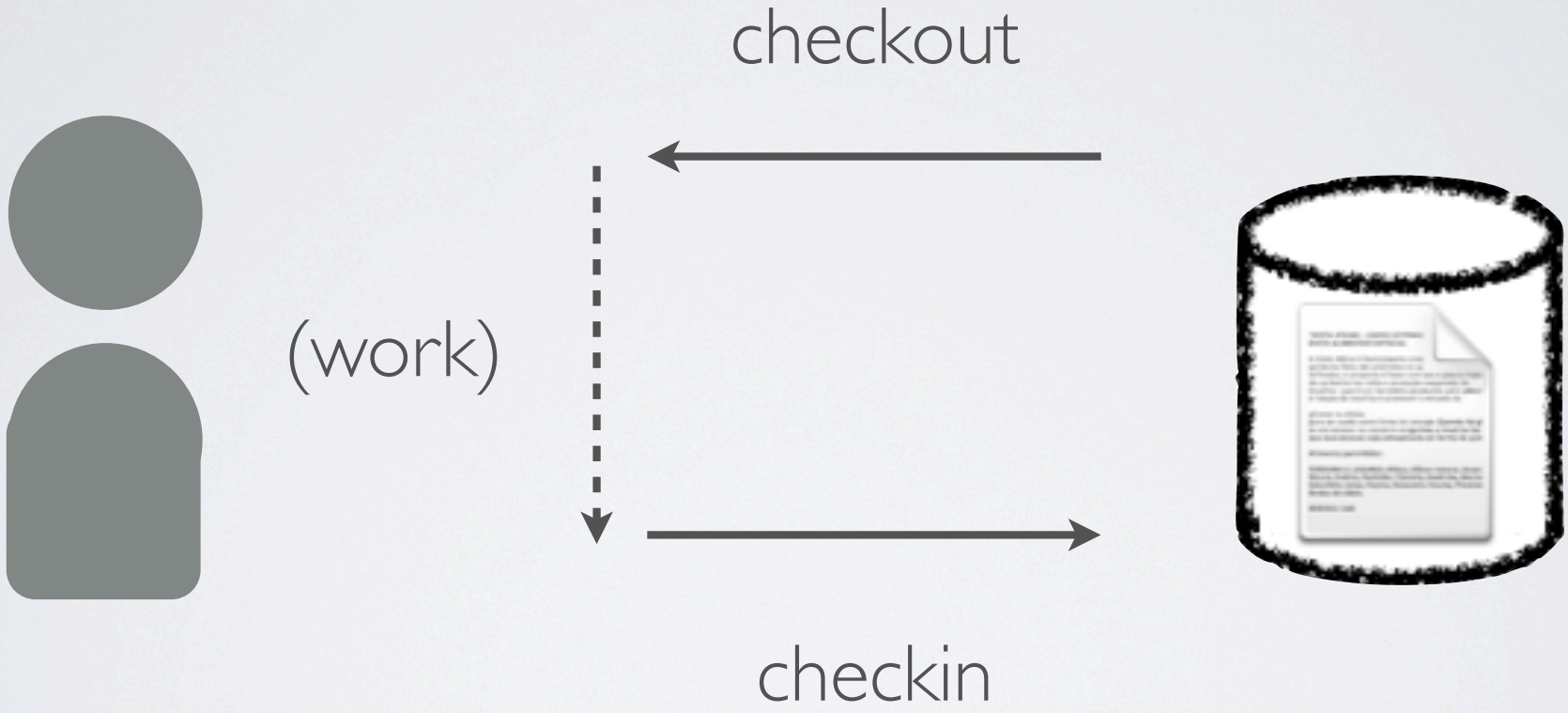


execution logs



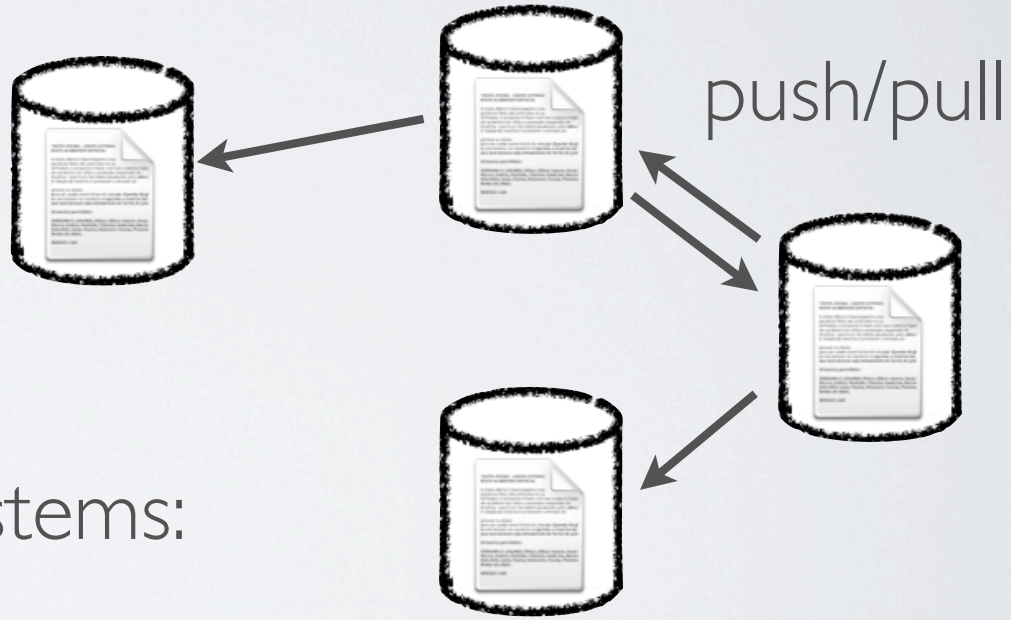
developer traces

# Changes to a software system are stored in the version control system



# Version control systems can be centralized or distributed

In a DVCS, **anyone** can fork a repository, and later bring back changes to the main repository.
There can be many more branches.




Common version control systems:
CVS↓, SVN→ (centralized)
Git↑ (distributed)

# Metadata in version control systems

Extract of an SVN log

```
18. r2852 | tyler | 2007-04-30 17:18:39
* guard_method: Made sure that we set the guard variable back to its old state *even if the bloc

M gemables/qualitysmith_extensions/Rakefile
A gemables/qualitysmith_extensions/doc_include/ReleaseNotes-0.0.28
M gemables/qualitysmith_extensions/lib/qualitysmith_extensions/module/guard_method.rb

------------------------------------------------------------------------
17. r2850 | tyler | 2007-04-30 16:44:08
* Changed Module#bool_attr_accessor to allow a state of nil

M gemables/qualitysmith_extensions/Rakefile
A gemables/qualitysmith_extensions/doc_include/ReleaseNotes-0.0.27
M gemables/qualitysmith_extensions/lib/qualitysmith_extensions/module/bool_attr_accessor.rb
```

Revision number
Author
Date
Comment

Files changed
Actual changes
(lines added/deleted)

# Issues affecting a software system are stored in an issue tracking system

Developers and managers use it to:

- discuss the reported defects
- gather additional information about defects
- detect duplicate bug reports
- prioritize which bug to fix first
- assign a bug to a given developer
- track the status of the bug

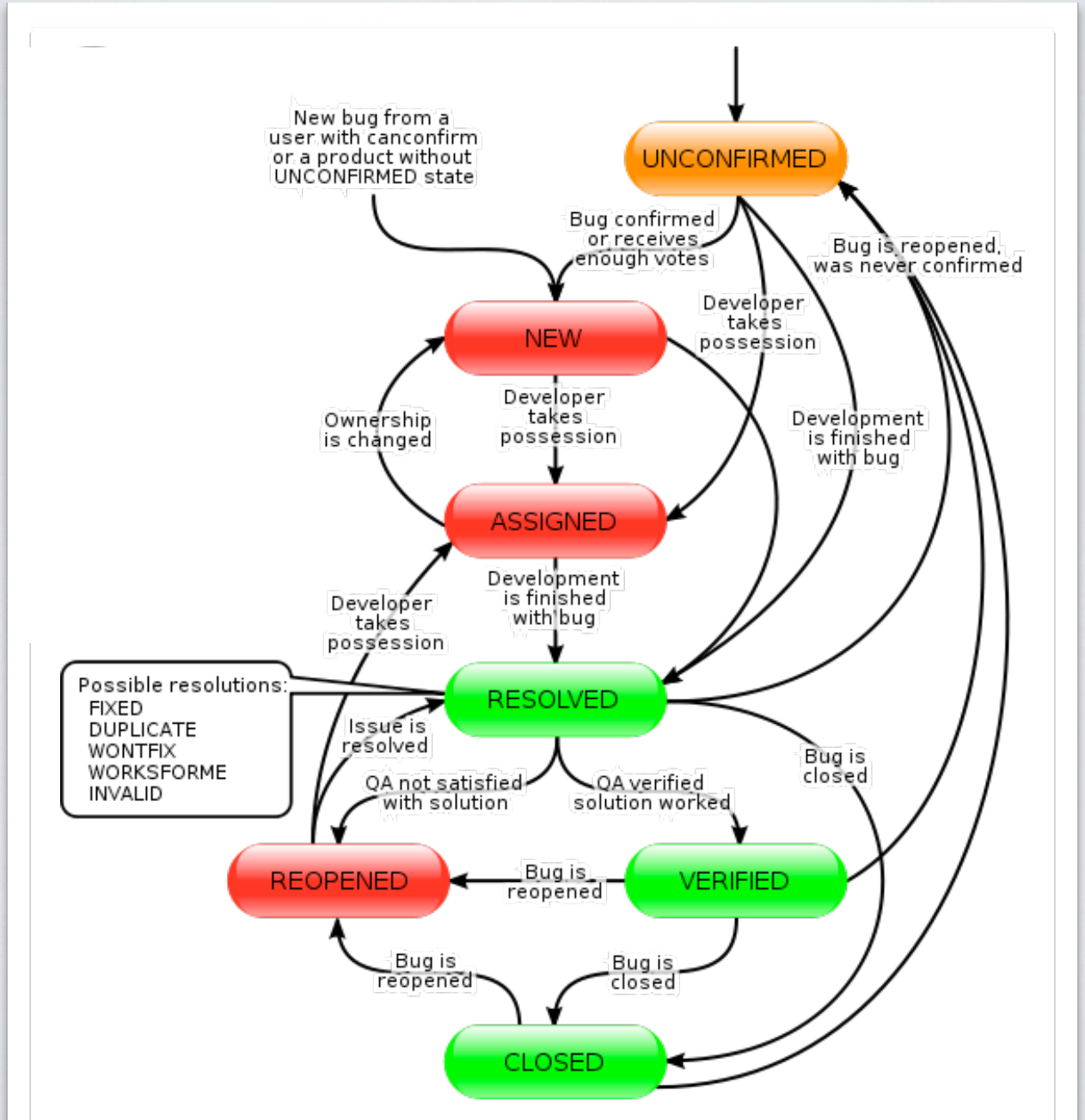
# Metadata in issue tracking systems

ID:
Product and Component:
Status and Resolution:
Assigned To:
Keywords:
Summary:
Version:

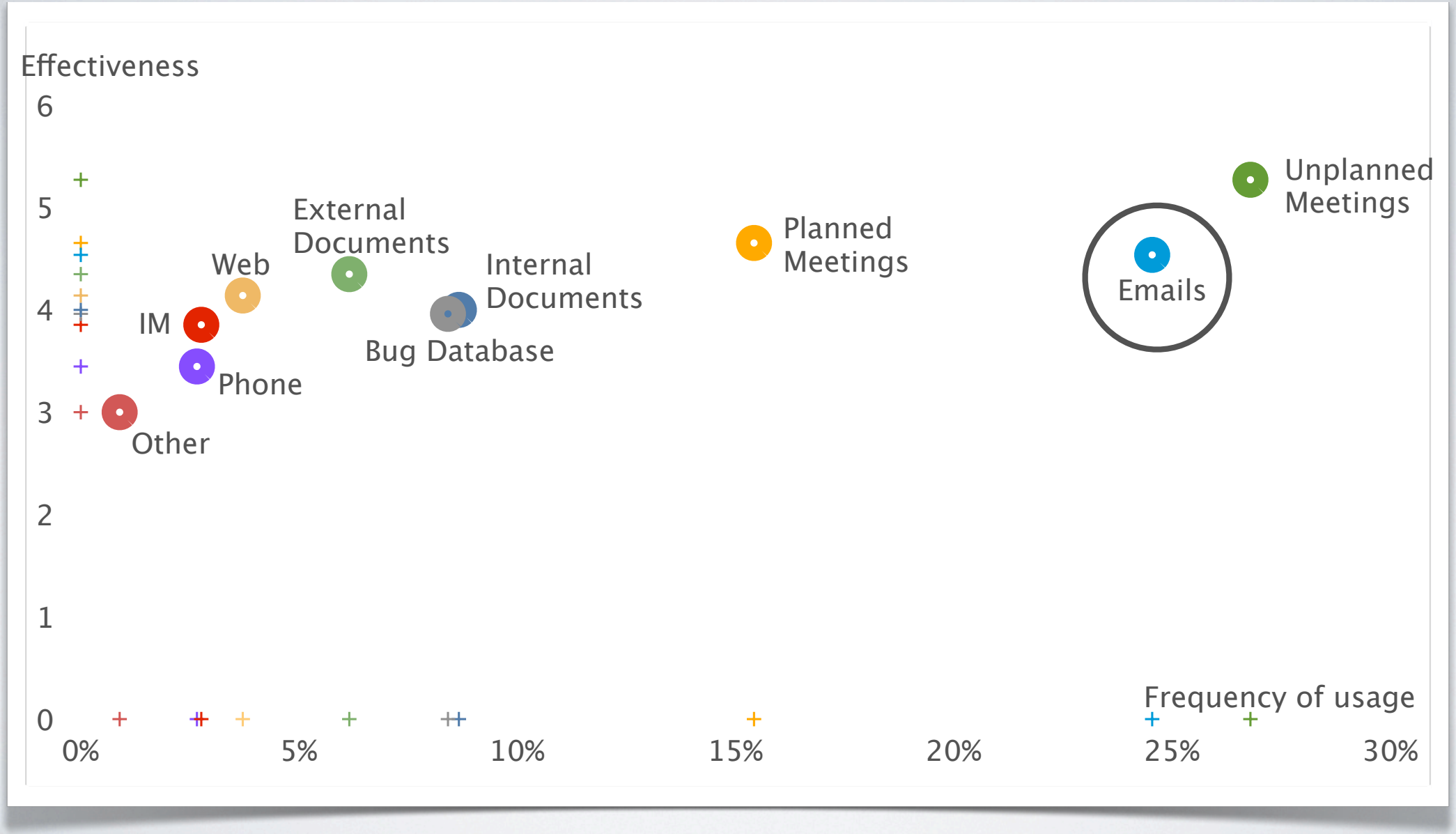
Priority:
Severity:
Target:
Reporter:
Attachments:
Dependencies:
Platform and OS:
Additional Comments:

Common bug tracking systems:
Bugzilla, Jira, Trac, Google code ...

# Possible statuses of a bug



# Emails are a widely used and effective means of communication



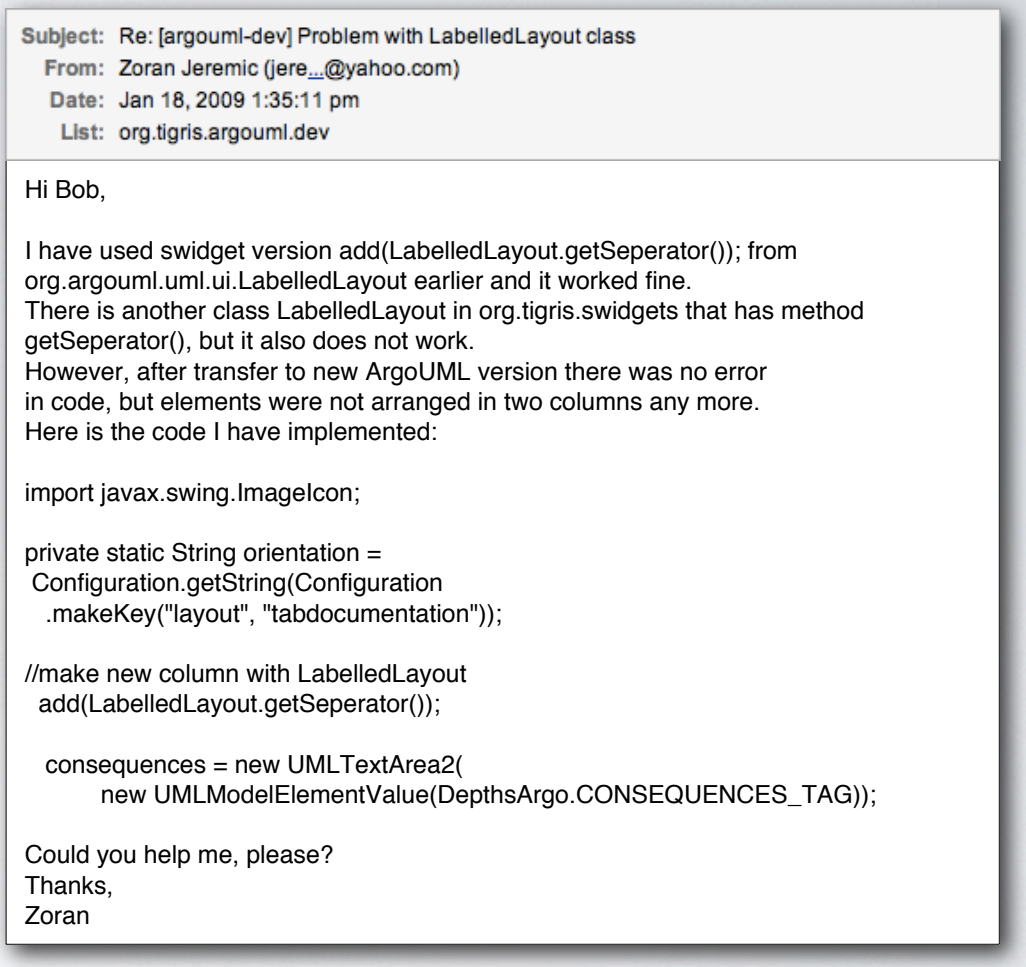

Maintaining mental models: a study of developer work habits
LaToza, Venolia, DeLine [ICSE 2006]

# Mailing lists are used to communicate between developers, or developers and users

Questions and answers by users

Discussions by developers on design, bugs and changes

Discussions contain mostly natural language, but also source code, stack traces, etc.



Subject: Re: [argouml-dev] Problem with LabelledLayout class
From: Zoran Jeremic (jere...@yahoo.com)
Date: Jan 18, 2009 1:35:11 pm
List: org.tigris.argouml.dev

Hi Bob,

I have used swidget version add(LabelledLayout.getSeperator()); from org.argouml.uml.ui.LabelledLayout earlier and it worked fine.
There is another class LabelledLayout in org.tigris.swidgets that has method getSeperator(), but it also does not work.
However, after transfer to new ArgoUML version there was no error in code, but elements were not arranged in two columns any more.
Here is the code I have implemented:

```
import javax.swing.ImageIcon;

private static String orientation =
 Configuration.getString(Configuration
   .makeKey("layout", "tabdocumentation"));

//make new column with LabelledLayout
  add(LabelledLayout.getSeperator());

   consequences = new UMLTextArea2(
         new UMLModelElementValue(DepthsArgo.CONSEQUENCES_TAG));
```

Could you help me, please?
Thanks,
Zoran

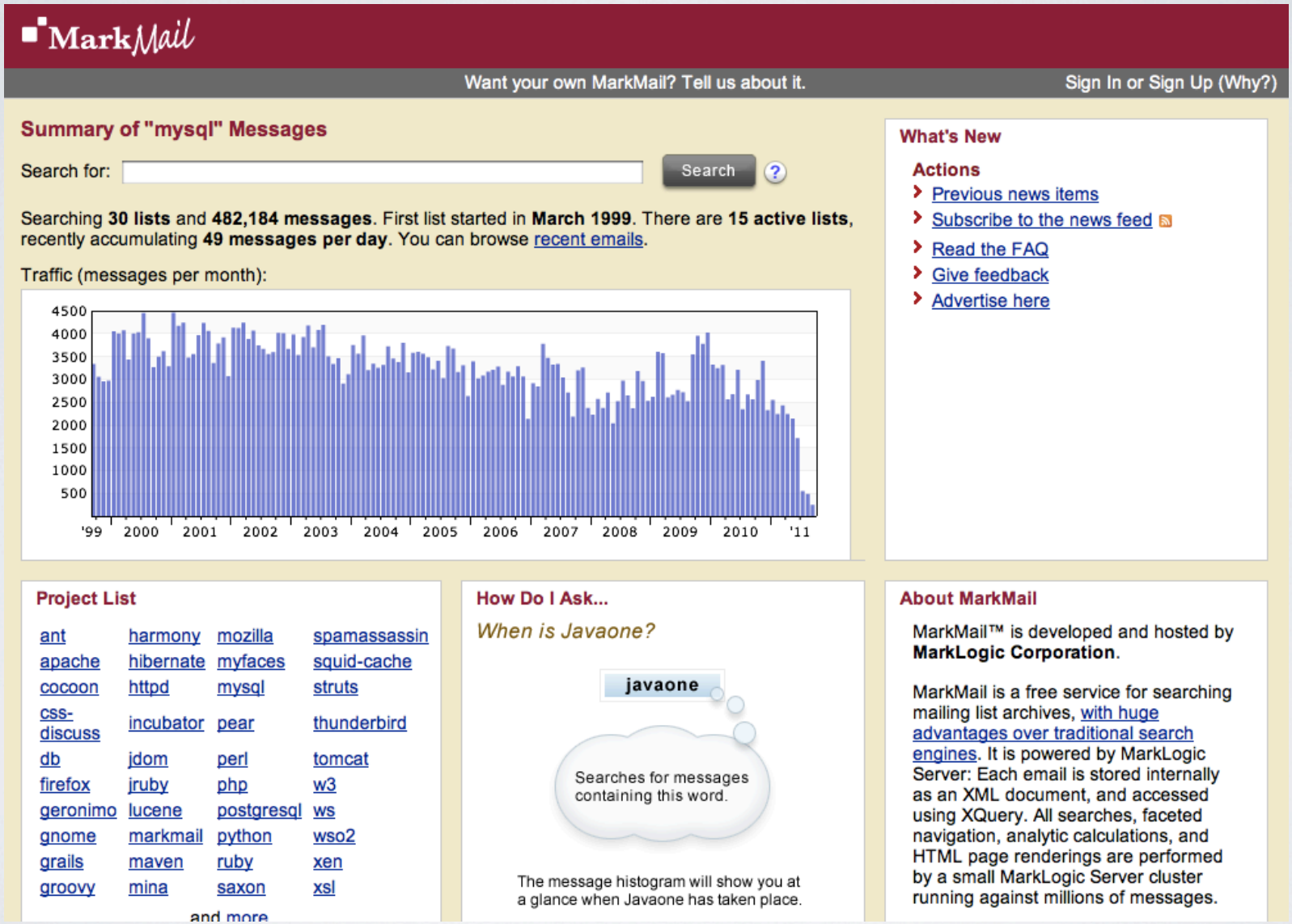
markmail.org is a crawlable search engine for mailing lists

# Metadata in mailing list archives

date
message id
subject
from
to
reply-to

contents
attachments

# markmail.org is a crawlable mailing list search engine




Other option: direct parsing of mbox files

# Other repositories

High-level models of systems
(class diagrams, sequence diagrams, etc)

Execution traces
(traces of runtime method calls/values)

Execution logs
(printouts, warning, errors from runtime executions)

Developer IDE traces
(navigation in the IDE, etc)

# Data pre-processing

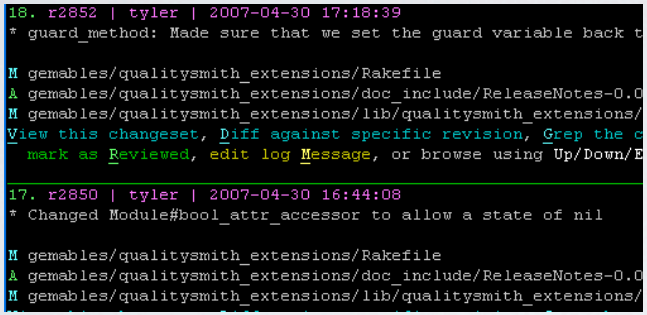
# The data in software repositories must be processed before being usable

Version control system: cleaning, parsing, linking

Issue tracker: parsing, linking to code

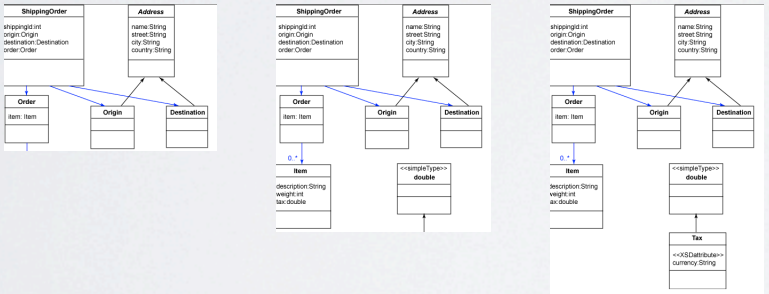
Mailing lists: parsing, linking

# Different levels of analysis

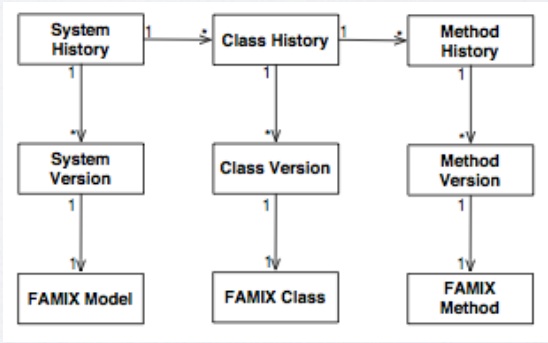



Change log only

Text file versions




Program versions



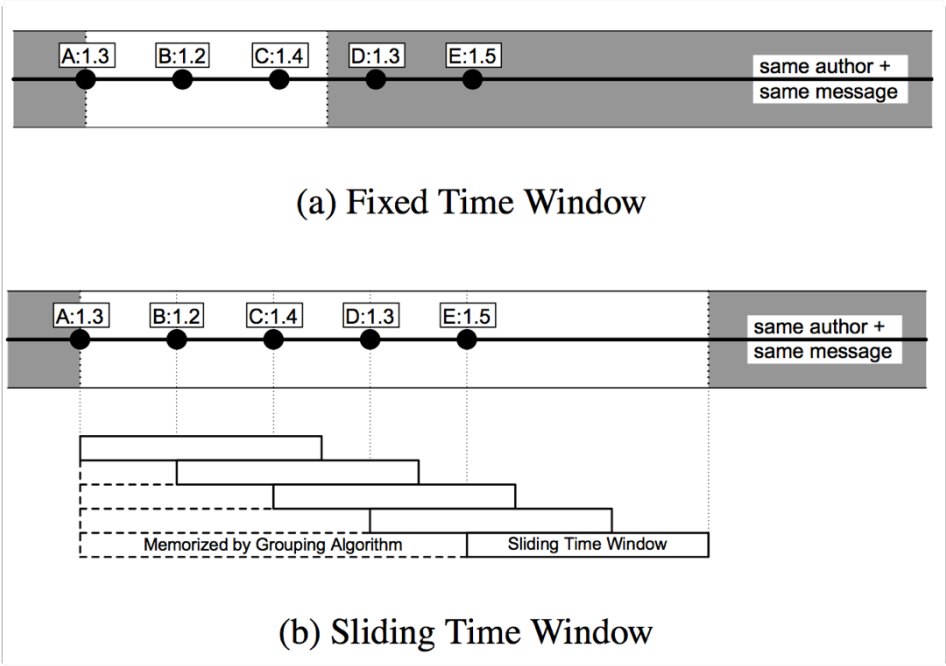
# Change log only analysis

- The change log must be parsed. The formats are well-documented, so this is not a problem.
- Get a local mirror of the repository you are interested in (e.g. with rsync, or dedicated tools)

# Processing the change log

- We retrieve the change log for the repository (e.g. cvs log, svn log --verbose)
- We get a list of revisions, with a time stamp, an author, a commit comment, etc.
- SVN revisions have the notions of changesets (files changed together); CVS does not.
- CVS revisions have the amount of lines added/deleted; SVN does not (use svn diff)

# Inferring transactions in CVS




- Heuristics: file committed by the same author, at roughly the same time, are in the same transaction.
- Threshold: 200 seconds

**Preprocessing CVS Data for Fine-Grained Analysis**

Thomas Zimmermann
Saarland University, Saarbrücken, Germany
tz@acm.org

Peter Weißgerber
Cath. Univ. of Eichstätt-Ingolstadt, Germany
peter.weissgerber@ku-eichstaett.de

# Modelling file versions

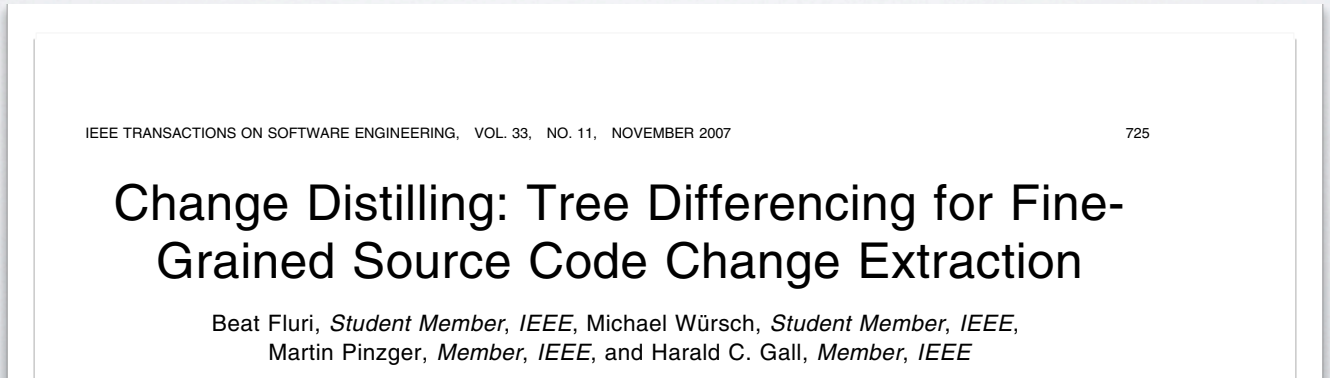
- Take snapshots of the code at regular interval (each version may be a lot), using cvs checkout or svn checkout
- Add information about the length of the source file (LOC, SLOC); compute lines added, removed
- Should one count blank lines and comment lines?
- Add information about directories (number of files present, etc)

# Parsing source code to build models

- Each revision of the system can be parsed
- We can compute metrics about each source code entity (NOM, McCabe, coupling metrics, clones, ...)
- This can be computationally intensive if there are a lot of versions. Sometimes they are *sampled*.
- Example tools: Moose and Churrasco, Kenyon, Evolyzer

# Linking and differencing versions

- Successive versions of file/entities can be linked, and evolutionary metrics can be computed (ENOM, ...)
- Accurate differences between versions can be computed (e.g., between v1 and v2, one **if** was added, and 4 calls to method **foo()** )
- ChangeDistiller can compute accurate differences for Java systems.



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 11, NOVEMBER 2007 725

Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction

Beat Fluri, *Student Member, IEEE*, Michael Würsch, *Student Member, IEEE*,
Martin Pinzger, *Member, IEEE*, and Harald C. Gall, *Member, IEEE*

# Preprocessing data from Git

- The overall scheme is the same, but there are a some differences.
- In a DVCS, getting a copy of the repository is the first step to do anything (git clone)

**The Promises and Perils of Mining Git**

Christian Bird*, Peter C. Rigby†, Earl T. Barr*, David J. Hamilton*, Daniel M. German†, Prem Devanbu*

*University of California, Davis, USA
†University of Victoria, Canada

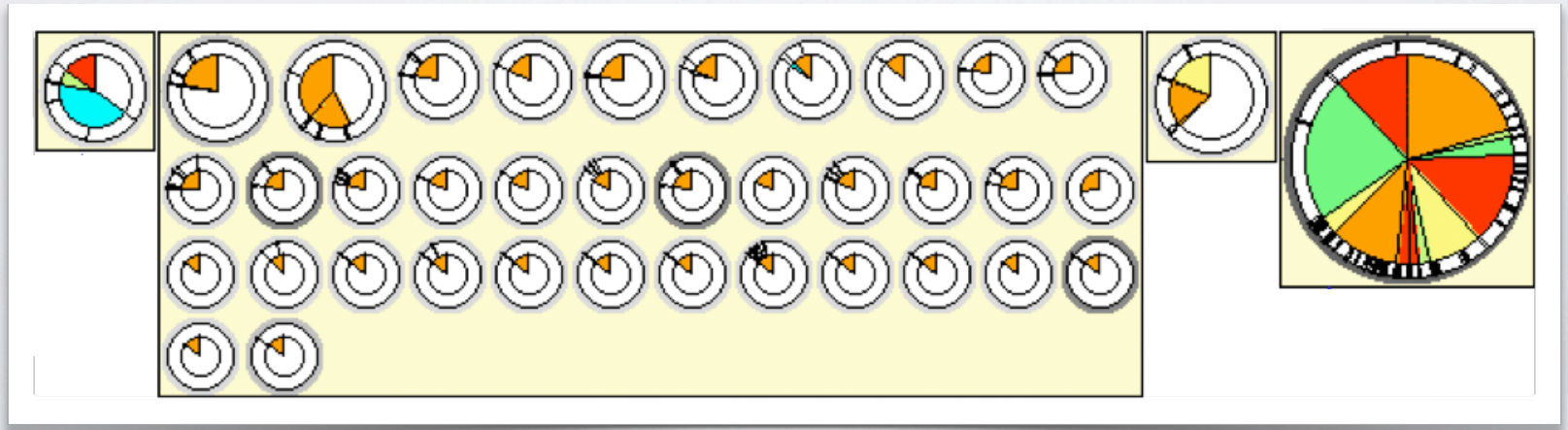
{bird,barr,hamiltod,devanbu}@cs.ucdavis.edu {pcr,dmg}@cs.uvic.ca

**Abstract**

*We are now witnessing the rapid growth of decentralized source code management (DSCM) systems, in which every developer has her own repository. DSCMs facilitate a style*

# Modelling levels for issue tracking systems

- Simple models with presence/absence of bugs for a given component and it's severity
- More complex models include:
  - history status of bugs
  - history of bug assignment
  - changes causing the bug and fixing it, etc



# How to gather the data depends on the bug tracker used

- e.g., in Bugzilla:
  - Download XML bug reports
  - Download attachments (if needed)
  - Download activities to build the history of the bug (if needed)

# Linking bugs and changes

- Base heuristic: If a bug is fixed, the developer will document it in the commit.
- Any number in a commit comment may be a revision number; look also for words such as "bug", "fix"
  - "fixed bug #12345"
- Check if the timestamp of the commit and the bug report closing are similar.
- Other issue tracking systems (e.g., Jira) feature a field in the bug tracker indicating related SCM commits.

# Detecting fixes to bugs

- The actual changes in the linked commit are assumed to be the fix to the bug referenced in the commit
- The lines of code can be extracted from the SCM system.

# Detecting bug-inducing changes

- Using cvs annotate, or svn blame, we can know who inserted the lines that were replaced in the bug fix (and when).
- We assume that these lines are the change that introduced the bug fix.

## When Do Changes Induce Fixes?

(On Fridays.)

Jacek Śliwerski
International Max Planck Research School
Max Planck Institute for Computer Science
Saarbrücken, Germany
sliwers@mpi-sb.mpg.de

Thomas Zimmermann Andreas Zeller
Department of Computer Science
Saarland University
Saarbrücken, Germany
{tz, zeller}@acm.org

**ABSTRACT**

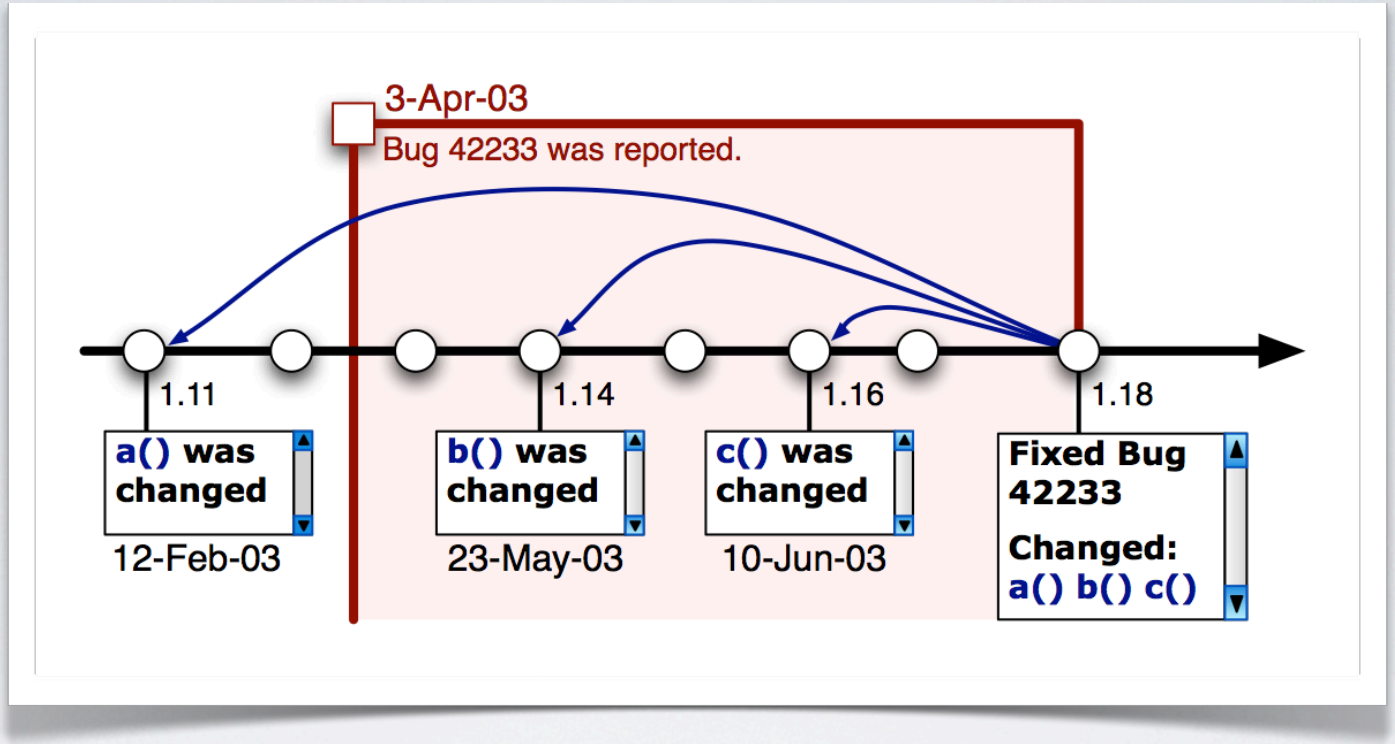
As a software system evolves, programmers make changes that sometimes cause problems. We analyze CVS archives for *fix-inducing changes*—changes that lead to problems, indicated by fixes. We show how to automatically locate fix-inducing changes by linking a version archive (such as CVS) to a bug database (such as BUGZILLA). In a first investigation of the MOZILLA and ECLIPSE history, it turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied.

**Which change properties may lead to problems?** We can investigate which properties of a change correlate with inducing fixes, for instance, changes made on a specific day or by a specific group of developers.

**How error-prone is my product?** We can assign a *metric* to the product—on average, how likely is it that a change induces a later fix?

**How can I filter out problematic changes?** When extracting the architecture via co-changes from a version archive, there is

# Filtering noise




Lines changed later than the bug report can not be part of the bug-introducing change (they may be partial fixes)

**When Do Changes Induce Fixes?**

(On Fridays.)

Jacek Śliwerski
International Max Planck Research School
Max Planck Institute for Computer Science
Saarbrücken, Germany

Thomas Zimmermann Andreas Zeller
Department of Computer Science
Saarland University
Saarbrücken, Germany

# Additional improvements

1. Use annotation graphs to provide more detailed annotation information
2. Ignore comment and blank line changes
3. Ignore format changes
4. Ignore outlier bug-fix revisions in which too many files were changed
5. Manually verify all hunks in the bug-fix changes

**Figure 2. Summary of approach**

**Automatic Identification of Bug-Introducing Changes**

Sunghun Kim[1], Thomas Zimmermann[2], Kai Pan[1], E. James Whitehead, Jr.[1]

[1]*University of California,*
*Santa Cruz, CA, USA*
*{hunkim, pankai, ejw}@cs.ucsc.edu*

[2]*Saarland University,*
*Saarbrücken, Germany*
*tz@acm.org*

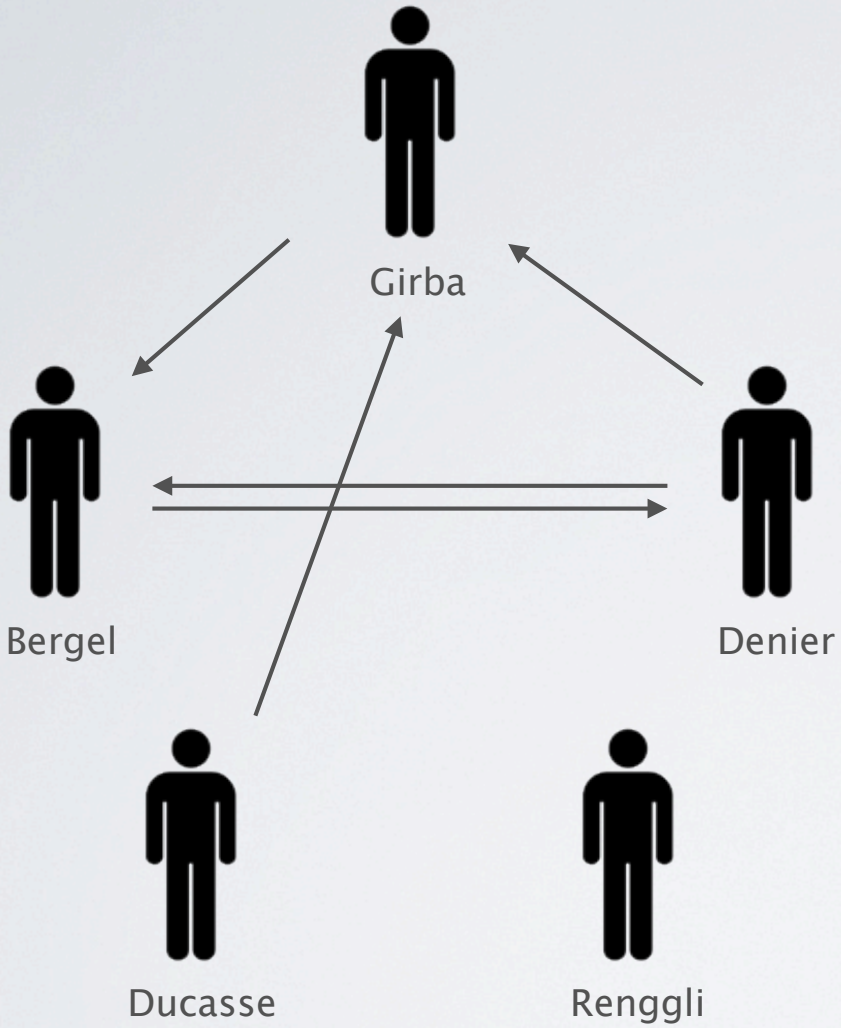
**Abstract**

*Bug-fixes are widely used for predicting bugs or finding risky parts of software. However, a bug-fix does not contain information about the change that initially introduced a bug. Such bug-introducing changes can help identify important properties of software bugs such as correlated factors or causalities. For example, they reveal which developers or what kinds of source code changes introduce more bugs. In contrast to bug-fixes that are relatively easy to obtain, the extraction of bug-introducing changes is challenging.*

permanently recording the change. As part of the commit, developers commonly (but not always) record in the SCM system change log the identifier of the bug report that was just fixed. We call this modification a *bug-fix change*.

Software evolution research leverages the history of changes and bug reports that accretes over time in SCM systems and bug tracking systems to improve our understanding of how a project has grown. It offers the possibility that by examining the history of changes made to a software project, we might better understand patterns of bug introduction, and raise developer awareness that

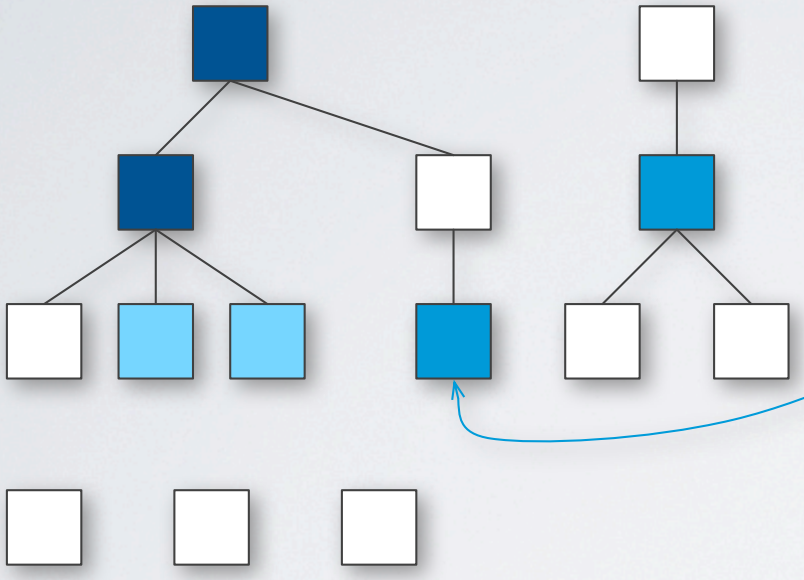
# Modelling mailing list data




| | |
|---|---|
| [Moose-dev] 2 more bugs in the MooseFinder | Alexandre Bergel |
| [Moose-dev] Re: 2 more bugs in the MooseFinder | Tudor Girba |
| [Moose-dev] Re: 2 more bugs in the MooseFinder | Simon Denier |
| [Moose-dev] Re: 2 more bugs in the MooseFinder | Alexandre Bergel |
| [Moose-dev] petit parser | Tudor Girba |
| [Moose-dev] Re: petit parser | Stéphane Ducasse |
| [Moose-dev] Re: petit parser | Simon Denier |
| [Moose-dev] Re: petit parser | Stéphane Ducasse |
| [Moose-dev] Re: petit parser | Alexandre Bergel |
| [Moose-dev] Re: petit parser | Tudor Girba |
| [Moose-dev] Re: petit parser | Alexandre Bergel |
| [Moose-dev] Re: petit parser | Lukas Renggli |
| [Moose-dev] Re: petit parser | Tudor Girba |
| [Moose-dev] Re: petit parser | Simon Denier |
| [Moose-dev] Re: petit parser | Stéphane Ducasse |
| [Moose-dev] Missing dependencies? | Alexandre Bergel |
| [Moose-dev] Re: Missing dependencies? | Simon Denier |
| [Moose-dev] Re: Missing dependencies? | Tudor Girba |
| [Moose-dev] Re: Missing dependencies? | Alexandre Bergel |
| [Moose-dev] Re: Missing dependencies? | Stéphane Ducasse |
| [Moose-dev] TheMooseBook? | Simon Denier |
| [Moose-dev] Re: TheMooseBook? | Tudor Girba |
| [Moose-dev] Fwd: [Pharo-project] Sprint at Bern | Stéphane Ducasse |

Basic level: time stamp, author, subjects, conversations

# Going further, we are interested in linking email and source code



| | |
|---|---|
| [Moose-dev] 2 more bugs in the MooseFinder | Alexandre Bergel |
| [Moose-dev] Re: 2 more bugs in the MooseFinder | Tudor Girba |
| [Moose-dev] Re: 2 more bugs in the MooseFinder | Simon Denier |
| [Moose-dev] Re: 2 more bugs in the MooseFinder | Alexandre Bergel |
| [Moose-dev] petit parser | Tudor Girba |
| [Moose-dev] Re: petit parser | Stéphane Ducasse |
| [Moose-dev] Re: petit parser | Simon Denier |
| [Moose-dev] Re: petit parser | Stéphane Ducasse |
| [Moose-dev] Re: petit parser | Alexandre Bergel |
| [Moose-dev] Re: petit parser | Tudor Girba |
| [Moose-dev] Re: petit parser | Alexandre Bergel |
| [Moose-dev] Re: petit parser | Lukas Renggli |
| [Moose-dev] Re: petit parser | Tudor Girba |
| [Moose-dev] Re: petit parser | Simon Denier |
| [Moose-dev] Re: petit parser | Stéphane Ducasse |
| [Moose-dev] Missing dependencies? | Alexandre Bergel |
| [Moose-dev] Re: Missing dependencies? | Simon Denier |
| [Moose-dev] Re: Missing dependencies? | Tudor Girba |
| [Moose-dev] Re: Missing dependencies? | Alexandre Bergel |
| [Moose-dev] Re: Missing dependencies? | Stéphane Ducasse |
| [Moose-dev] TheMooseBook? | Simon Denier |
| [Moose-dev] Re: TheMooseBook? | Tudor Girba |
| [Moose-dev] Fwd: [Pharo-project] Sprint at Bern | Stéphane Ducasse |

# Linking emails to source code entities

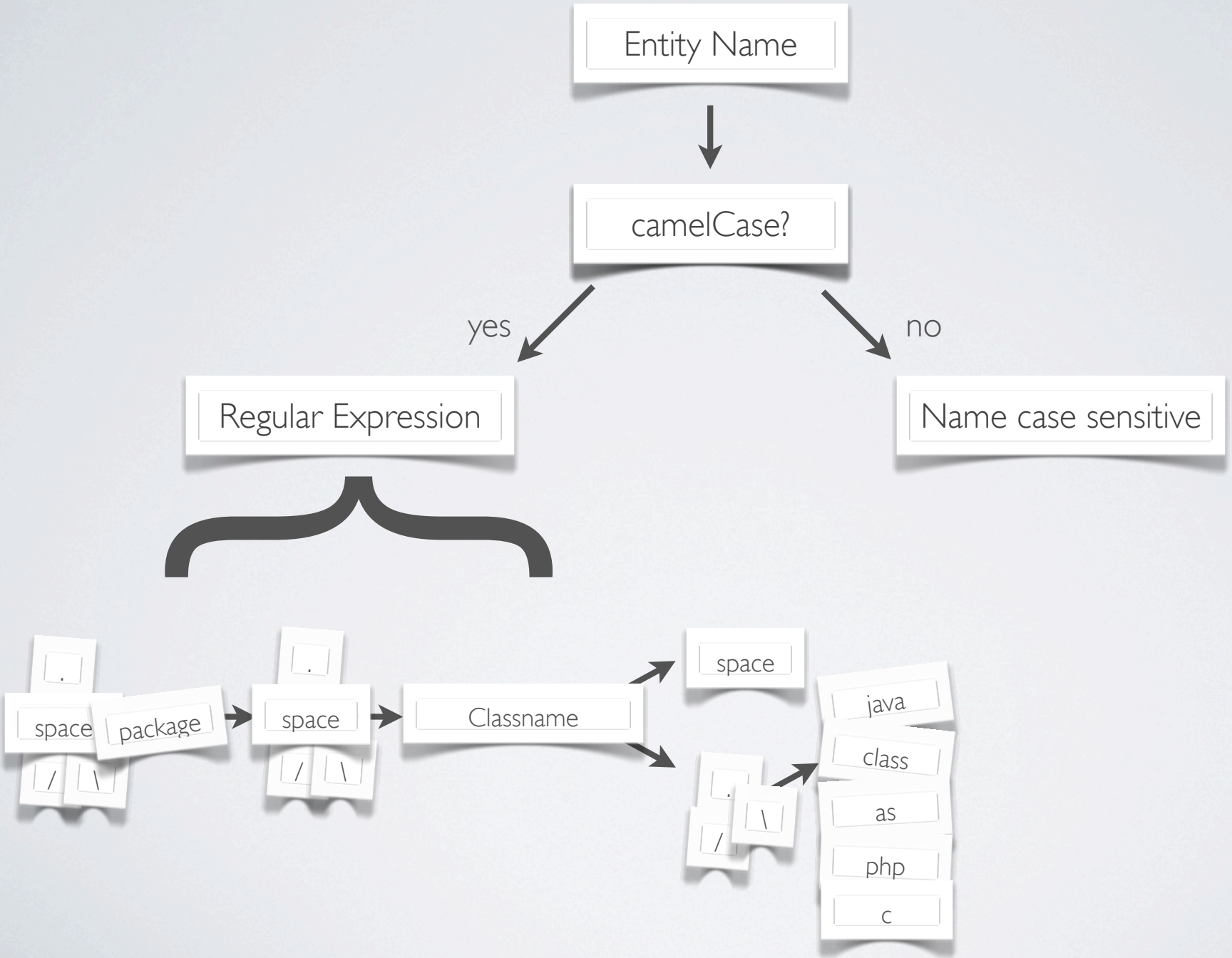
- Several heuristics exist:
  - If a class name is mentioned, assumes it is the class. This fails for simple names such as "Table"
  - Be more strict on the format of entity name: e.g, "org.foo.bar.Table"
  - Use information retrieval techniques (LSI, LDA).

**Linking E-Mails and Source Code Artifacts**

Alberto Bacchelli, Michele Lanza
REVEAL @ Faculty of Informatics
University of Lugano
{alberto.bacchelli,michele.lanza}@usi.ch

Romain Robbes
PLEIAD @ DCC & REVEAL
University of Chile & University of Lugano
rrobbes@dcc.uchile.cl

# Most accurate linking technique so far:



# Going further, one can identify source code fragments in emails

Subject: Re: [argouml-dev] Problem with LabelledLayout class
From: Zoran Jeremic (jere...@yahoo.com)
Date: Jan 18, 2009 1:35:11 pm
List: org.tigris.argouml.dev

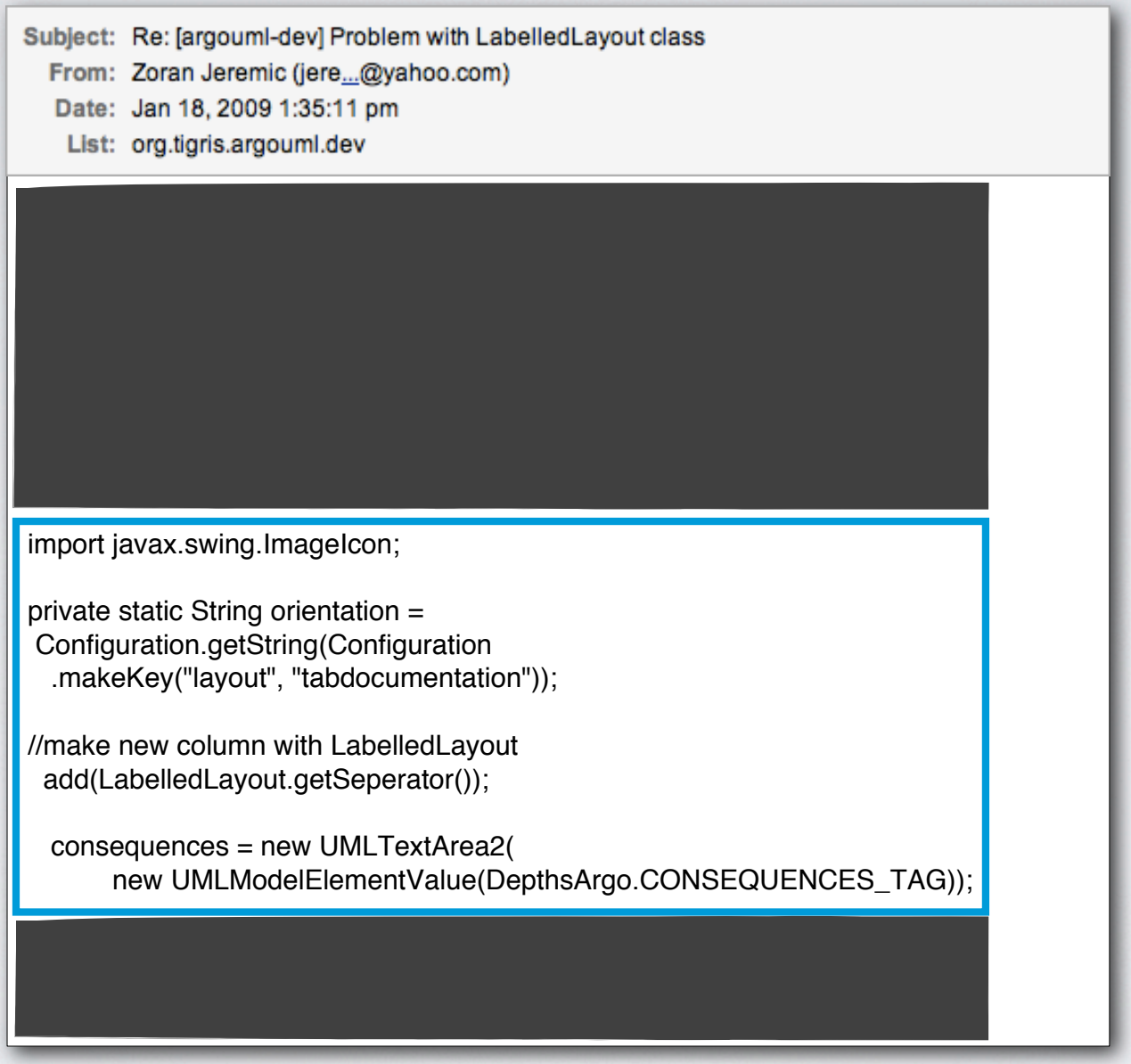
Hi Bob,

I have used swidget version add(LabelledLayout.getSeperator()); from org.argouml.uml.ui.LabelledLayout earlier and it worked fine.
There is another class LabelledLayout in org.tigris.swidgets that has method getSeperator(), but it also does not work.
However, after transfer to new ArgoUML version there was no error in code, but elements were not arranged in two columns any more.
Here is the code I have implemented:

```
import javax.swing.ImageIcon;

private static String orientation =
 Configuration.getString(Configuration
   .makeKey("layout", "tabdocumentation"));

//make new column with LabelledLayout
  add(LabelledLayout.getSeperator());

   consequences = new UMLTextArea2(
         new UMLModelElementValue(DepthsArgo.CONSEQUENCES_TAG));
```

Could you help me, please?
Thanks,
Zoran

# Going further, one can identify source code fragments in emails



Subject: Re: [argouml-dev] Problem with LabelledLayout class
From: Zoran Jeremic (jere...@yahoo.com)
Date: Jan 18, 2009 1:35:11 pm
List: org.tigris.argouml.dev

```
import javax.swing.ImageIcon;

private static String orientation =
 Configuration.getString(Configuration
   .makeKey("layout", "tabdocumentation"));

//make new column with LabelledLayout
  add(LabelledLayout.getSeperator());

   consequences = new UMLTextArea2(
          new UMLModelElementValue(DepthsArgo.CONSEQUENCES_TAG));
```

# Recap

# Summary

- MSR has it roots in the very first empirical studies of software engineering
- The "paper trail" left by tools used by developers to manage evolving software contain useful data
- Since the last 15 years and the advent of open-source software, MSR is a thriving research area.
- There are many software repositories to mine from: changes, bugs, emails, IDE traces, and others

# Version control systems

- The data in version control systems needs to be preprocessed
  - parsing change logs
  - recovering transactions
  - origin analysis
  - computing diffs between versions
  - modelling individual versions

# Issue tracking systems

- Defect data also needs to be processed:
  - retrieving and parsing defect logs
  - linking defect to changes
  - finding bug fixes
  - finding bug-introducing changes

# Mailing list archives

- Necessary processing
  - retrieving mail data
  - parsing threads, author information, etc
  - linking emails to source code
  - identifying source code fragments