Change Analysis with Evolizer and ChangeDistiller

Harald C. Gall, Beat Fluri, and Martin Pinzger, *University of Zurich*

Evolizer, a platform for mining software archives, and ChangeDistiller, a change extraction and analysis tool, enable the retrospective analysis of a software system's evolution.

Software must undergo continuous change or it becomes progressively less useful.[1] Many software systems represent business processes that must be adapted continuously owing to changing environments, business reorientation, or modernization.

To understand why a software system becomes less evolvable when it undergoes continuous change, and to reduce its maintenance costs, we investigate its change history and obtain knowledge to support its evolution. Our research field, *software evolution analysis*, is the retrospective analysis of software systems' evolution, or history. Such evolution comprises all phases and activities in the software system's life cycle.

A software system's historical data has two dimensions.[2] The *what and why dimension* focuses on understanding the software evolution phenomenon—that is, it tries to answer why making continuous changes to software increases its complexity. The *how dimension* focuses on supporting developers and project managers in their daily business—for instance, by providing feedback during development.

In this article, we discuss the potential of mining software archives by presenting the results of three experimental studies and a tool that supports software evolution in integrated development environments (IDEs). Our techniques and tools focus on change analysis, or discovering all kinds of change types, from interface to condition and method invocation changes. We also describe how such a fine-grained change analysis works. Evolizer, our platform for mining software archives, is the basis for ChangeDistiller, our change extraction and analysis tool, which investigates fine-grained source code changes. While coarse-grained change analysis is limited to the level of files and textual differences, fine-grained change analysis provides detailed information on the level of statement and declaration changes.

The Evolizer Software Evolution Analysis Platform

We developed Evolizer, a platform to enable software evolution analysis, in Eclipse (www.eclipse.org). It's similar to Kenyon[3] or eROSE,[4] but it systematically integrates change history with version and bug data. Evolizer provides a set of metamodels to represent software project data along with adequate importer tools to obtain this data from software project repositories. Our current implementation provides support for importing and representing data from the versioning control systems CVS (Concurrent Versions System) and SVN (Subversion), the bug-tracking system Bugzilla, Java source code, and fine-grained source code changes, as well as the integration of these models. Using the Eclipse plug-in extension facilities and the Hibernate object-relational mapping framework (www.

 0740-7459/08/$25.00 © 2009 IEEE

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on November 16, 2009 at 04:23 from IEEE Xplore. Restrictions apply.

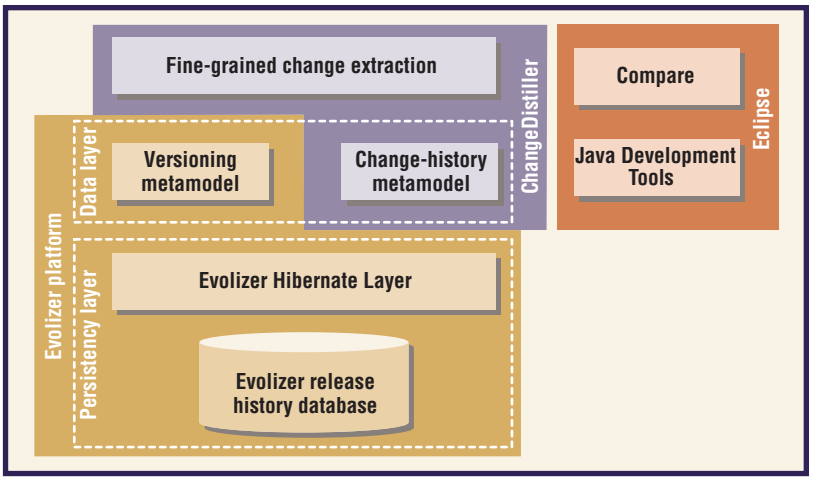


Figure 1. Evolizer's core architecture. It comprises the Evolizer platform, which integrates various data sources for software evolution analysis, and ChangeDistiller, which extracts and analyzes source code changes.

hibernate.org), extending existing metamodels and data importers in Evolizer, or adding new ones, is straightforward. Models are defined by Java classes, annotated with Hibernate tags, and added to the list of model classes. Evolizer loads this list of classes and provides it to the other Evolizer plug-ins for accessing the software evolution data. Using the Eclipse plug-in mechanism with extensible metamodels is Evolizer's main advantage over existing mining tools.

Figure 1 depicts Evolizer's core architecture. The Evolizer release history database (RHDB) stores a software system's extracted historical data, which includes files, revisions, modification reports, and author information. On top of the data and persistency layers, analysis tools such as ChangeDistiller can access all the evolution data that Evolizer provides.

Source Code Change Extraction with ChangeDistiller

Source code can be represented by its abstract syntax tree (AST). Eclipse ships with Java Development Tools (JDTs), which provide a rich set of functionality to create and manipulate ASTs of Java source files. Our change-distilling algorithm uses tree differencing on the ASTs of two subsequent versions of a particular class.[5] The algorithm calculates an *edit script* that contains basic tree edit operations and transforms the older AST into the newer AST. We use the basic tree edit operations—`insert`, `delete`, `move`, and `update`—on AST nodes.

Our taxonomy of source code changes defines change types according to tree edit operations in the AST. We use the taxonomy to translate an edit script into concrete source code changes. In addition, the taxonomy defines the change significance level, which expresses the possible impact a change type might have on other source code entities and whether it might alter the functionality. We use change significance levels to measure how relevant each particular source code change would be.

Currently, our taxonomy defines more than 40 change types for source code entities.[6] Table 1 shows an excerpt of our current change type taxonomy. We divide these change types into body-part and declaration-part categories of attributes, classes, and methods. Each change type obtains a

Table 1

Change types and significance levels[6]

Change type	Significance
Body-part change types	
Conditions	
Loop condition	Medium
Control structure condition	Medium
Else-part insert	Medium
Else-part delete	Medium
Statements	
Statement insert/delete	Low
Statement ordering change	Low
Statement parent change	Medium
Statement update	Low
Comments	
Comment insert/delete	None
Comment update	None
Declaration-part change types	
Classes and interfaces	
Class insert/delete	Crucial
Class update	Crucial
Interface insert/delete	Crucial
Interface update	Crucial
Parameters	
Parameter insert/delete	Crucial
Parameter ordering change	Crucial
Parameter type change	Crucial
Parameter renaming	Medium
Return types	
Return type insert/delete	Crucial
Return type update	Crucial

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on November 16, 2009 at 04:23 from IEEE Xplore. Restrictions apply.

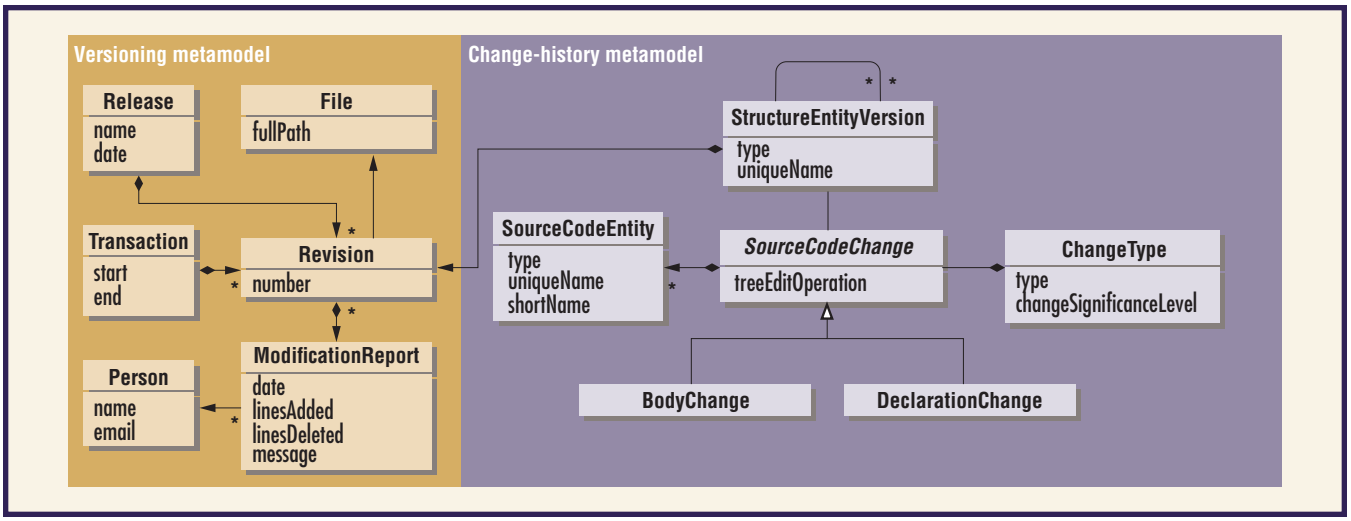


Figure 2. Evolizer versioning and change history metamodels. We integrate the two metamodels representing versioning and source code change data via the Revision entity.

change significance level of `none`, `low`, `medium`, `high`, or `crucial`. For certain change types, the change significance level adapts to a source code entity's accessibility modifier. For instance, a return type change of a public method has a higher change significance level than that of a private method. We aim to differentiate less relevant changes from significant changes—for instance, changes that impact functionality.

Leveraging the information provided by ASTs gives us precise information about a source code change. In addition to the information that a particular source code entity has changed, tree edit operations also provide information about where the change occurred. For instance, we can tell that a method invocation `fred.bar()` was moved into an if-statement with the condition `fred != null`.

ChangeDistiller works with Java and is built on top of the Evolizer platform to mine source code changes in software archives. It plugs our change history metamodel into the Evolizer persistency layer to integrate versioning with source code change data. Figure 2 depicts the integrated versioning and change history metamodels. The versioning metamodel represents source files with all their revisions and modification reports. Each revision links to the changed classes, methods, and attributes in the corresponding file revision. For each version of a structure entity, we reference a list of source code changes. We categorize source code changes as body and declaration changes. A distilled change type then holds a reference to the changed piece of code—that is, nodes in the AST.

Extracting source code changes with Evolizer and ChangeDistiller is straightforward.

1. We use the Eclipse CVS plug-in to check out the `HEAD` revision or any other release of a CVS repository holding a Java project.
2. We configure Evolizer to use a MySQL database.
3. We use the Evolizer CVS importer plug-in to obtain each Java source file's version history and store it to the Evolizer RHDB.
4. We use ChangeDistiller to retrieve the source code changes between each pair of subsequent revisions of classes, methods, and attributes.

The result is an RHDB that contains the versioning history of all Java source files and the detailed source code changes for each revision.

Analyzing Source Code Changes

Our experiments with Evolizer and ChangeDistiller contributed insights for understanding software evolution, particularly in commenting behavior, change type patterns, and changes that fixed bugs.

Coevolution of Comments and Code

We studied in detail the following software systems: ArgoUML, Azureus, 43 Eclipse plug-ins (Core, JDT, PDE, and so forth), jEdit, JFreeChart, and a commercial Java Web framework. We also investigated whether—and under which circumstances—comments and code coevolve. We developed an approach to associate comment change types with source code entities and conducted three experiments to study coevolution questions.[7]

Experiment 1. First, we investigated how the relative growth rate of source code and comments evolves. We checked whether the same relative amount of

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on November 16, 2009 at 04:23 from IEEE Xplore. Restrictions apply.

code and comments are added over time. We expected that over time, the ratio of comments and source code evolve similarly.

We found that the relative ratio of source code to comments remains stable over time in all the software systems we investigated. This doesn't necessarily mean that newly added code is commented well; on the contrary, half of the investigated systems had less than 50 percent of their source code commented.

Experiment 2. Next, we examined whether adding comments depends on the source code entity. We assumed that particular source code entities are commented more than others.

We found that whether or not a source code entity gets commented depends highly on its type. There's also a partial order in the likeliness of which entity gets commented: classes are more often commented than methods and attributes; if-statements and loops are more often commented than method invocations or other statements.

Experiment 3. Finally, we looked into whether comments are adapted when source code is changed (that is, whether comments are kept up-to-date) and when the adaptations occur. We assumed that redocumentation is an integral part of the software development process, and we further assumed that programmers often neglect to adapt comments when changing source code.

Our results showed that in six out of eight investigated systems, code and associated comments were cochanged in the same revision 90 percent of the time. Although API comments weren't changed in the same revision, they were redocumented later. Also, source code changes induced more than 50 percent of comment changes.

Quality analysis. We can use our findings to qualify a software system's commenting process. Our tools automatically generate the data for all three experiments and store them in the Evolizer RHDB. Historical data and change types are available for tool access, filtering (per class, package, or change type, such as comment change), or further analysis. Thus, conventional statistical tools can import and analyze the data by first merging data and then generating corresponding plots such as histograms or correlations. As a result, for each experiment, we consolidated comment change data and interpreted the data as follows.

Proportion of code to comments. Comparing the growth factor of the number of commented and noncommented lines of code shows whether the proportion of comment lines to code lines increased, decreased, or remained stable over a software system's history. This doesn't indicate that the source code is well commented or that the comments are meaningful, but it does show whether a system's developers comment their code consistently over time.

Comment quality. The proportion of code to comments indicates the amount of comments in a software system. Computing growth and proportion is straightforward and can be done on the fly with modern IDEs. But simply counting the lines of code and the comment lines hides two major aspects: it counts dead code as comment lines, and it doesn't consider which source code entity types are commented. Therefore, we complement the results and filter out dead code because it harms our understanding of source code. The type of source code entity commented—and the extent of the comments—affects the quality. The less dead code present, and the more that declaration parts and scopes are commented, the better the comments' quality and the system's maturity.

Up-to-date comments. We can assess whether comments are kept up-to-date or at least adapted in revisions after the associated source code entity has changed. This shows whether redocumentation is integral to the development process. For instance, we experienced that redocumentation for declaration parts was major in four of the investigated software systems. The sooner the comments are adapted to source code changes, the better we rate a system's commenting process. But redocumentation is also positive because source code comments are added—better late than never.

Change Type Patterns

Certain source code changes are mostly applied together. For instance, a parameter renaming impacts all statements that access the parameter inside the method body. These statements must be adapted to the parameter change.

We assume that a recurring coding activity is reflected in the same specific group of change types. Therefore, we aim to identify those groups of change types that frequently appear together and to describe such groups' semantics by a change type pattern. To achieve those goals, we apply agglomerative hierarchical clustering on change type vectors. A change type vector denotes the set of change types of a method revision obtained from the Evolizer RHDB.[8]

The type of source code entity commented—and the extent of the comments—affects the quality.

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on November 16, 2009 at 04:23 from IEEE Xplore. Restrictions apply.

We applied the cluster analysis on jEdit, JFreeChart, and a commercial Java Web framework. The experiments showed that change type patterns reveal differences in exception flow usage and that certain control flow changes indicate change efforts to make the code consistent to coding guidelines. We found several change type patterns.

Developers used exception flow to check the parameter values for certain conditions. For example, in the method

```
public void setCategory(Comparable category) {
  this.category = category;
}
```

they inserted a parameter check:

```
public void setCategory(Comparable category) {
  if (category == null) {
    throw new IllegalArgumentException(
      "Null `category' argument.");
  }
  this.category = category;
}
```

Developers also switched from the multiple- to the single-exit principle. For instance, the method

```
public boolean hasAnyNextSteps() {
  if (getStep() instanceof SpStep) {
    return true;
  }
  return conf().hasAnyNextSteps(get());
}
```

was restructured to

```
public boolean hasAnyNextSteps() {
  boolean result = false;
  if (getStep() instanceof SpStep) {
    result = true;
  } else {
    result = conf().hasAnyNextSteps(get());
  }
  return result;
}
```

Developers swapped the conditional branches in If statements. For instance, the method

```
public void print() {
  DocFile docFile = getDocFile();
  if (selectPrinter != null) {
    OM.getInst().print(docFile, selectPrinter);
  } else {
    OM.getInst().print(docFile);
  }
}
```

was restructured to

```
public void print() {
  DocFile docFile = getDocFile();
  if (selectPrinter == null) {
    OM.getInst().print(docFile);
  } else {
    OM.getInst().print(docFile, selectPrinter);
  }
}
```

Moreover, certain API convention cleanups spread over a software system's whole history.

For the population of various change type patterns in a specific software system, we automated our approach up to the interpretation of the change type clusters. By specifying a cluster threshold, we can fully automate the extraction of groups of frequently appearing change types. To make the identified pattern specific, we just have to confirm the suggested change type patterns in ChangeDistiller. Using the specified change type patterns, we can then search for specific and similar occurrences automatically.

The identified change type patterns can be leveraged for two specific scenarios.

Consistency of changes. Discovering change type patterns lets us perform a consistency analysis of the source code changes, especially when paradigm shifts occur. For example, the introduction of the single-exit principle in the Web framework started in a specific period and should have been implemented consistently in all parts of the architecture. With ChangeDistiller, we can discover this point in time and find violations of this principle. We can, therefore, leverage our approach to inform developers of inconsistent changes.

Feedback during evolution. Most change type patterns can be seen as code cleanup changes, so developers might argue that they aren't exciting and obviously happen during software development. But we can also learn from these patterns: Either coding guidelines are adapted frequently or they're not followed strictly. Revealing inconsistencies in applying coding guidelines is an important part of software quality assurance. We can provide feedback during evolution with a recommender that can be configured either by users or by learning from the

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on November 16, 2009 at 04:23 from IEEE Xplore. Restrictions apply.

occurred change type patterns. Moreover, a recommender notifies programmers, who are new to a software project, of certain guidelines and supports their fast adoption and correct usage.

Method Invocation Changes

Like Sunghun Kim, Kai Pan, and James Whitehead,[9] we were curious about finding change patterns that fix bugs. From these change patterns, we can learn how to avoid or fix recurring bugs. We focused on method invocations and obtained corresponding changes from the Evolizer RHDB. Next, we postprocessed them to do the following:

- Classify the changes into bug fixes and normal changes. Changes whose revision references a bug number in the commit message are bug fixes; all other changes are normal.
- Aggregate changes of invocations of the same method. For each method invocation, the method signature is resolved and the invocation changes are aggregated to the corresponding signature.
- Extract patterns from among the changes. A pattern appears when we can extract similarities between the single changes. For instance, when an if-statement with a certain condition is often put around a method invocation, the corresponding changes form a pattern.

We obtained promising results by analyzing the method invocation changes in the Eclipse project. Interestingly, method invocations to the JDK library changed most often and were involved in many bug fixes. We also could extract patterns among the changes. For instance, in our experiment with the Eclipse project, a significant amount of if-statements with `<qualifier>.contains(<argument>)` as the condition were put around `List.add(Object)` method calls. We aim to avoid future bugs by leveraging these change patterns. To achieve this goal, we're extending our tools to provide automated support.

Potential Evolution Support in IDEs

The results of our experiments motivated us to develop tools that ease software systems' evolution. Integrating such tools into Eclipse provides a closed feedback loop:

- historical data of specific development processes are collected in the development environment,
- empirical approaches to analyze this data will be automated, and
- rules as well as recommendations will emerge from this data to effectively support developers.[10]

To show how such an integration will support developers in their daily business, we describe our recommendation tool. Figure 3 shows a screen shot of our current prototype in Eclipse. The tool is built on top of Evolizer and leverages method invocation changes extracted by ChangeDistiller. For a detailed discussion of this research prototype, see our previous research.[11]

In our experiment with method invocation changes, we observed that a significant number of bugs are fixed with similar source code changes. We aim at providing additional change suggestions to reduce the number of future bugs. For that, we've developed a tool that suggests changes when a developer inserts a certain method invocation.

The prototype is integrated into Eclipse's incremental project builder to generate recommendations for a particular class during development. Our tool aims to provide instant feedback on suitable recommendations using visual representations familiar to developers and supported by the majority of IDEs.

Assume the following scenario depicted in Figure 3. Using the Eclipse IDE, a developer is performing a feature change for which she must modify the method `resolveClassPath(..)` in the class `JavaProject` in the plug-in org.eclipse.jdt.core. She adds the call `resolvedEntries.add(rawEntry)`. By inspecting the call and code fragment in which the invocation is inserted, the tool queries the Evolizer RHDB to fetch method invocation change patterns for the added call. If patterns are available, it marks the corresponding line and highlights the method invocation statement (see Figure 3, Part 1). This approach integrates seamlessly with the existing views provided by Eclipse because annotations added by the tool are additionally listed in the problems view (see Figure 3, Part 2).

To apply a recommended change, we give the developer a list of quick fixes (see Figure 3, Part 3). The tool suggests those changes that were applied frequently and were often involved in bug fixes. In the case of the `resolvedEntries.add(rawEntry)` method invocation, there were two suggestions:

- Add an If statement with the condition `resolvedEntries != null`, and call the method inside the Then part of the If statement.
- Add an If statement with the condition `!resolvedEntries.contains(rawEntry))`, and call the method inside the Then part of the If statement.

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on November 16, 2009 at 04:23 from IEEE Xplore. Restrictions apply.

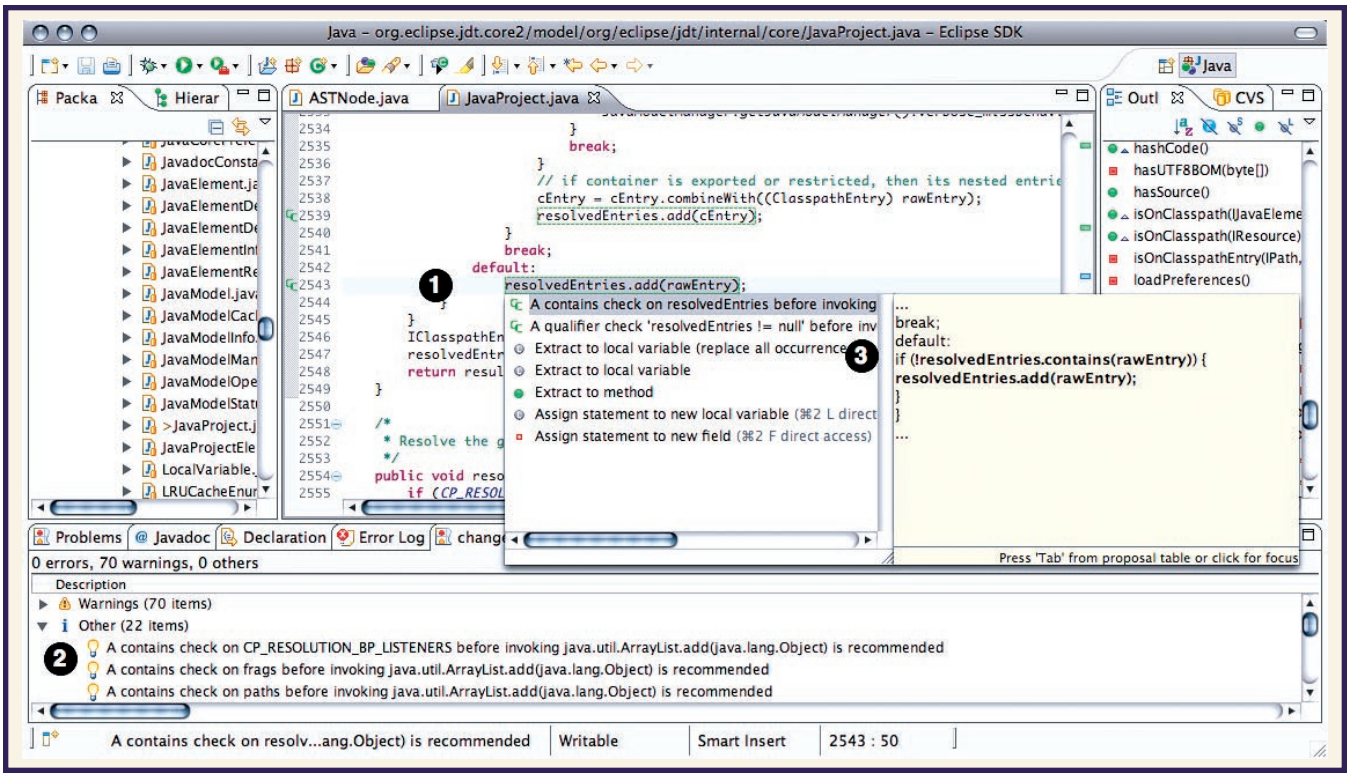


Figure 3. Method invocation change recommendation tool in action. Eclipse's quick fix feature provides recommendations for adding argument and qualifier checks before method invocations. (1) After the developer adds a call, the tool queries Evolizer to fetch change patterns for the call, marks the corresponding line, and highlights the invocation statement. (2) Eclipse adds the annotated invocations to the problems view. (3) The tool gives the developer a list of quick fixes.

Selecting a quick fix instantly performs the modifications to the AST needed for applying a change. For example, in the Eclipse code base, several bug fixes with respect to the `List.add(Object)` method added a null dereferencing check for the list object and a check for whether the list already contains the object.

Ongoing work also foresees the integration of comment and code coevolution data to suggest appropriate comment adaptation to code changes and the integration of change type patterns to support consistent changes.

Software evolution analysis aims at retrospectively analyzing software systems' history to understand software evolution and reduce maintenance costs. Typically, software archives such as source code version-control and issue-tracking systems offer historical data. Several approaches leverage this information, but existing techniques suffer from the coarse-grained information available for source code changes (for example, source lines added or lines deleted).

On the basis of our findings in changes that fixed bugs, we developed a new prototype tool that, given an actual source code change, points the developer to potential pitfalls and recommends effective fixes. Our tool presents a promising example of how fine-grained source code change information can support evolution. This motivates us to deepen our knowledge of software evolution phenomena and provide further evolution support in our ongoing and future work.

References

1. M.M. Lehman, "Programs, Life Cycles and Laws of Software Evolution," *Proc. IEEE*, vol. 68, no. 9, 1980, pp. 1060–1076.
2. T. Mens and S. Demeyer, eds., *Software Evolution*, Springer, 2008.
3. J. Bevan et al., "Facilitating Software Evolution Research with Kenyon," *Proc. Joint 10th European Software Eng. Conf. and the 13th ACM SIGSOFT Symp. Foundations of Software Eng.*, ACM Press, 2005, pp. 177–186.
4. T. Zimmermann et al., "Mining Version Histories to Guide Software Changes," *IEEE Trans. Software Eng.*, vol. 31, no. 6, 2005, pp. 429–445.
5. B. Fluri et al., "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Trans. Software Eng.*, vol. 33, no. 11, 2007, pp. 725–743.

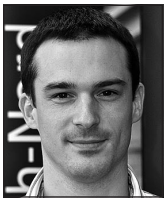
Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on November 16, 2009 at 04:23 from IEEE Xplore. Restrictions apply.

6. B. Fluri and H.C. Gall, "Classifying Change Types for Qualifying Change Couplings," *Proc. 9th Int'l Conf. Program Comprehension*, IEEE CS Press, 2006, pp. 35–45.
7. B. Fluri, M. Würsch, and H.C. Gall, "Do Code and Comments Co-evolve? On the Relation between Source Code and Comment Changes," *Proc. 14th Working Conf. Reverse Eng.*, IEEE CS Press, 2007, pp. 70–79.
8. B. Fluri, E. Giger, and H.C. Gall, "Discovering Patterns of Change Types," *Proc. 23rd IEEE/ACM Int'l Conf. Automated Software Eng.*, IEEE CS Press, 2008, pp. 463–466.
9. S. Kim, K. Pan, and E.J. Whitehead, "Memories of Bug Fixes," *Proc. 14th ACM SIGSOFT Symp. Foundations Software Eng.*, ACM Press, 2006, pp. 35–45.
10. A. Zeller, "The Future of Programming Environments: Integration, Synergy, and Assistance," *Proc. Future of Software Eng.*, IEEE CS Press, 2007, pp. 316–325.
11. B. Fluri, J. Zuberbühler, and H.C. Gall, "Recommending Method Invocation Context Changes," *Proc. 1st Int'l Workshop Recommender Systems for Software Eng.*, ACM Press, 2008; http://pages.cpsc.ucalgary.ca/~zimmerth/rsse-2008/papers/p1-fluri.pdf.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

About the Authors

Harald C. Gall is a professor of software engineering in the University of Zurich's Department of Informatics. His research interests include software evolution, software quality analysis, software architecture, reengineering, collaborative software engineering, and service-centric software systems. Gall received his PhD in informatics from the Vienna University of Technology. He's also a program cochair of the 2011 International Conference on Software Engineering. Contact him at gall@ifi.uzh.ch; http://seal.ifi.uzh.ch/gall.



Beat Fluri is a senior research associate in the Software Engineering Group in the University of Zurich's Department of Informatics. His main research interest is software evolution, focusing on source code change analysis and recommender systems. Fluri received his PhD in informatics from the University of Zurich. He's a member of the IEEE, the IEEE Computer Society, the ACM, and the ACM Sigsoft. Contact him at fluri@ifi.uzh.ch; http://seal.ifi.uzh.ch/fluri.

Martin Pinzger is an assistant professor in software engineering in the Department of Software Technology at the Delft University of Technology. When he was working on this article, he was a senior research associate in the Software Engineering Group in the University of Zurich's Department of Informatics. His research interests are in software engineering, focusing on software evolution analysis and software design and quality analysis. Pinzger received his PhD in informatics from the Vienna University of Technology. He's a member of the IEEE, the IEEE Computer Society, and the ACM. Contact him at pinzger@ifi.uzh.ch; seal.ifi.uzh.ch/pinzger.

Give Your Career a **Boost**

IEEE computer society CSDA Certified Software Development Associate

IEEE computer society CSDP Certified Software Development Professional

"Having the CSDP helped me make the case for strengthening our software quality process, which drastically reduced our production support costs by 40%."

Phanindra Mankale, CSDP
F500 Manufacturing Company

In today's environment, strengthening your resume is more important than ever. Whether you are an entry-level or mid-career software practitioner, we have the answer:

Distinguish yourself with one of the IEEE Computer Society's software development credentials.

Stand out from the others with the **CSDA/CSDP**

For more information, and to see how these credentials have helped other practitioners, go to: www.computer.org/getcertified

Authorized licensed use limited to: Technische Universiteit Delft. Downloaded on November 16, 2009 at 04:23 from IEEE Xplore. Restrictions apply.