SpyWare: A Change-Aware Development Toolset

Romain Robbes
Faculty of Informatics, University of Lugano
romain.robbes@lu.unisi.ch

Michele Lanza
Faculty of Informatics, University of Lugano
michele.lanza@unisi.ch

ABSTRACT

Our research is driven by the motivation that *change* must be put in the center, if one wants to understand the complex processes of software evolution. We built a toolset named SpyWare which, using a monitoring plug-in for integrated development environments (IDEs), tracks the changes that a developer performs on a program *as they happen*. SpyWare stores these first-class changes in a change repository and offers a plethora of productivity-enhancing IDE extensions to exploit the recorded information.

1. INTRODUCTION

The only constant in software is that it changes: Software must be continuously tailored to fit new or updated requirements. This has been formulated in Lehman's first law of software evolution [4], which states that a software system must be continuously adapted, or become less and less useful. Given this situation, it is surprising that most development tools (with the exception of versioning systems) still consider a software system as its source code only, disregarding any available historical information. This information, when available in the form of versioning system archives, has been proven to be useful for several applications [5], [11] and fostered the quickly growing community dedicated to the mining of software repositories [3, 1]. However, the evolutionary information mostly used, versioning system archives (primarily from systems such as CVS and Subversion), suffers from several problems (detailed in [6], [7]), such as incomplete data sources and noise.

With SpyWare our goal was to start from a clean slate by capturing changes *as they happen* at the level of program entities rather than files and lines. For instance, our format allows us to describe refactorings. Instead of relying on the developer to save changes, our tool chain monitors changes as they happen in the IDE to have a finer granularity of changes. This allows us to model the change history as a sequence of first-class change operations, rather than a sequence of successive versions of the program. Since we use the IDE as an information source, we exploit the integration with the IDE at its maximum, by providing our enhancements as IDE extensions rather than stand-alone tools. More than creating single tools, we thus created the SpyWare platform, which allows us to:

- monitor developer activity as it happens in the IDE, using a monitoring plugin,
- convert the changes the developer did to the program to first-class change operations,
- stores the changes in a repository for later use,
- record higher-level changes such as refactorings,
- generate, by executing change operations, part or the whole of the system at any point in time,
- access the change history of any package, class, instance variable, method, or statement defined in the system,
- show the differences between two states of the program, using color-coding to reflect the type of changes,
- measure the extent of changes using structural metrics and the type of changes that were performed,
- visualize how the system was changed with metrics, graphs, and interactive visualizations,
- generalize concrete changes to the system to reusable program transformations,
- access all of its functionality from the IDE, rather than a stand-alone tool, to ease its usage.

In the following, we describe the approach and the change model we defined, before detailing the SpyWare toolset.

2. CHANGE-BASED SOFTWARE EVOLUTION IN A NUTSHELL

SpyWare is based on our work on *Change-Based Software Evolution* (CBSE) [7], whose aim is to accurately model how software evolves by treating *change* as a first-class entity.

The Model. We model software evolution as a sequence of changes that take a system from one state [1] to the next by means of semantic (i.e., non text-based) transformations. These transformations are inferred from the activity recorded by the event notification system of IDEs such as Eclipse, whenever the developer incrementally modifies the system. Examples are the modification of the body of a method or a class, but also higher-level changes offered by refactoring engines. In short, we do not view the history of a software system as a sequence of versions, but as the sum of *change operations* which brought the system to its actual state.

[1] We define the state as being the source code of the program, as opposed to the dynamic run-time state

ICSE'08, May 10–18 2008, Leipzig, Germany.

Atomic Change Operations

Creation	creates a node *n* for an entity of a given type *t*
Addition	adds a node *n* as a child of a given parent *p*
Removal	removes a node *n* from the childs of its parent *p*
Property change	changes value *v* of property *p* of node *n*
Insertion	inserts node *n* at location *m* as a child of (method) parent *p*.
Deletion	deletes a node *n* from location *m* in the children of the given parent *p*.

Composite Change Operations

Developer action	A unit of change from a developer's viewpoint, e.g., changing the definition of a class, adding or changing a method. It can contain several atomic changes, e.g., a method addition contains a creation, an addition and a name change for itself, and contains all the changes involved in the creation and the addition of each body statement
Refactoring	A behavior-preserving automatic code transformation [2], e.g., the *rename method* refactoring involves changing a method's name, and all references from the old method name to the new name. These developer actions can be grouped into a higher-level entity representing the refactoring itself
Development session	It aggregates all changes (refactorings or developer actions) done during a single development session by a developer. We used it in evolution analysis, as it is the equivalent in size to a versioning system commit. We provide more details on sessions in [8]

Table 1: Change operations supported by Spyware.

Program Representation. We represent programs as domain-specific entities rather than text files. Since we focus on object-oriented programs, we store and analyse constructs such as classes and methods, not files and lines. Thus, a software system is an evolving abstract syntax tree (AST) containing nodes which represent packages, classes, methods, variables and statements. A node *a* is a child of a node *b* if *a* contains *b*, e.g., a package contains classes. Nodes have *properties*, which vary depending on the node type. For a class we can have its name and superclass; for a method its name, return type and access modifier; for a variable its name, type and access modifier, etc. The name of an entity is a property since we provide identity with unique identifiers. Each AST entity has a *change history* containing all changes applied to it during the system's evolution.

Change Operations. They represent the evolution of the system under study: Change operations are the actions a developer performs when he changes a program, which in our model are captured and reified. They represent the transition from one state of the evolving system to the next. Change operations are *executable*: A change operation *c* applied to the state *n* of the program yields the state *n+1* of the program. Some examples of change operations are: adding/removing classes/methods to/from the system, changing the implementation of a method, or refactorings [2]. We support *atomic* and *composite* change operations (detailed in Table 1):

Atomic Change Operations. Atomic changes are, at the finest level, operations on the program's AST. Atomic change operations are executable, and can be undone: An atomic change contains all the necessary information to update the model by itself, and to compute its opposite atomic change (to allow undo). By iterating on the list of changes we can generate all the states the program went through during its evolution.

Composite Change Operations. While atomic changes model the evolution of programs, the finest level of granularity is not always the best suited. Representing the entire evolution of a system only by its atomic modifications leads to an overwhelming mass of information. Hence we abstract change operations into higher-level *composite changes*, e.g., moving a class from package *A* to package *B* consists in removing it from *A* and adding it to *B*. These two atomic changes can be grouped in a single *move class* change.

The Change Repository. Changes are captured as they happen in the IDE, through the SpyWare monitoring plugin, which tracks the developer's actions. It reacts to them by querying the IDE for the information it needs to model the developer's actions as change operations and stores them in a change repository, where they are used by a plethora of tools described in the next section.

3. TOOLS BUILT ON TOP OF SPYWARE

On top of the change-recording and storing facility we described above, we built several tools. These tools, some of which are shown in Figure 1, are:

The Launcher (1). It is the starting point to use several of the tools included in spyware. The launcher allows us to load the model of a system in the environment. Beyond that, it allows us to choose the model we should work on if several of those are loaded, and launch other tools, such as metric graphs, change lists, and view browsers.

The Metric Graph (2). It follows the evolution of one or more structural metrics (i.e., number of classes, number of methods, average size of methods) on one or more projects. The metric graph is interactive and allows zooming, as well as summoning one or more view browsers at any point in time.

The Change List. Lists all the changes done to a part or the whole system, in a hierarchy from sessions, refactorings, developer-level actions, up to atomic changes. The change list supports simple queries to filter the changes it displays.

The View Browser (3). It shows the code of the system at a specific date. Multiple browsers can be open concurrently at several distinct dates of the evolution of the system, allowing comparisons between several versions of the system. These browsers feature forward and back buttons to advance or go backwards in time.

The Change Matrix (4). It is an interactive visualization showing the changes made to the system across time (on the x axis), and several entities (on the y axis). By default, the change matrix shows the changes to all classes and methods in the system, but it can be restrained to a part of the system. Additionaly, it can show changes to packages and classes rather than classes and methods. In all cases, entities are laid out in the order they have been created. Developer-level changes are shown as colored squares on the figure, according to the type of change that was performed. They can be expanded to show the evolution of the size of the entity they modified. This visualization allows one to grasp the evolution of the design of a system at a glance. Several patterns, such as god classes, data classes, or methods being too tightly coupled (when they are systemically modified together), can be detected with the change matrix.

The Session Browser (5). This tool presents all the coding sessions of the system visually. The history of the system is summed up in the smallest space possible thanks to a compact figure, the session sparkline. Inspired by Tufte's concept, a sparkline is a word-sized, high density graphic. Each of the sparklines encode the length of the session (with its length), as well as the type (with colors) and intensity (with the length of the bars) of activity that occured through it. Each sparkline is interactive, and can show the changes performed in it with tooltips. Sessions can be searched for the use of a word. Change-based metrics on each sessions are shown on the right and allow to compare session with each other. If a session is deemed interesting, it can be inspected with the session inspector.

The Session Inspector (6). The session inspector focuses on a single development session. It allows each change in the session to be replayed. The tool can show the state of an entity before and after each change using text highlighting depending on the nature of changes performed. The tool is able to identify that, for instance, a single variable was added to a method, rather than just telling that a variable was changed. Its usage is described in detail in [8].

The Software Animator (7). Currently a stand-alone tool, this tool replays the history of a system not as a single visualization, but as an interactive movie. Packages, classes and methods are shown as 3D objects which grow, reduce or blink when they are changed. The speed of the playback can be set from real-time to making hours or days pass in a few seconds. The animation can be paused, rotated, zoomed, and each entity can be queried to show extra information.

The Change Factory. The Change Factory is a tool which uses the change history to ease future changes to the system. Based on a concrete change performed previously on the system, the programmer can use the Change Factory to create a reusable generic program transformation which can be applied in other contexts afterwards. The Change Factory allows concrete entities to be converted to variables in the transformations, and allows to insert higher-level control structures such as loops and conditionals to make the transformation more general.

The Optimist Completer. Still under development, this tool uses the change history of a system to make the code completion tool of the IDE more accurate. By default, code completion tools return matches alphabetically. The Optimist Completer reorders the matches based on the usage of the entities to make the most likely to be used match reach the top of the list.

4. TOOL INFORMATION

Experience. The SpyWare toolset has been used in a variety of contexts. We used the View Browser, the Metric Graphs and the Change Matrix to help us understand the development of 9 student projects [7, 9]. We also used the Session Browser and the Session Inspector to browse more than 300 development sessions of SpyWare itself, as well as more than 100 development sessions from a third-party project [8]. The ability offered by the toolset to seamlessly move from one level of detail to the other allows the user to both gain a high-level view of the system, and understand the evolution of single entities.

Implementation. SpyWare has been developed over the last 3 years as part of a PhD thesis. It has been extensively used in a number of case studies and can thus be considered a mature prototype. SpyWare currently features two implementations, one in Squeak Smalltalk, and one in Java as an Eclipse plugin [10]. The prototype has not been optimized for performance or space, but both parameters are acceptable on a current laptop (2.4Ghz).

Tool Availability. SpyWare can be obtained at `http://www.inf.unisi.ch/phd/robbes/spyware.html`

The Software Animator can be found at: `http://atelier.inf.unisi.ch/~garciaal/SA/index.html`

5. REFERENCES

[1] S. Diehl, H. Gall, and A. E. Hassan, editors. *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*. ACM, 2006.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[3] A. E. Hassan, R. C. Holt, and S. Diehl, editors. *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*. ACM, 2005.

[4] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[5] V. B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, pages 296–305. ACM Press, 2005.

[6] R. Robbes and M. Lanza. Versioning systems for evolution research. In *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pages 155–164. IEEE CS Press, 2005.

[7] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:93–109, Jan. 2007.

[8] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 155–164, 2007.

[9] R. Robbes, M. Lanza, and M. Lungu. An approach to software evolution based on semantic change. In *Proceedings of FASE 2007 (10th International Conference on Fundamental Approaches to Software Engineering)*, pages 27–41, 2007.

[10] Y. Sharon. Eclipseye - spying on eclipse. Bachelor's thesis, University of Lugano, June 2007.

[11] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: raising risk awareness. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, pages 107–110, New York, NY, USA, 2005. ACM Press.

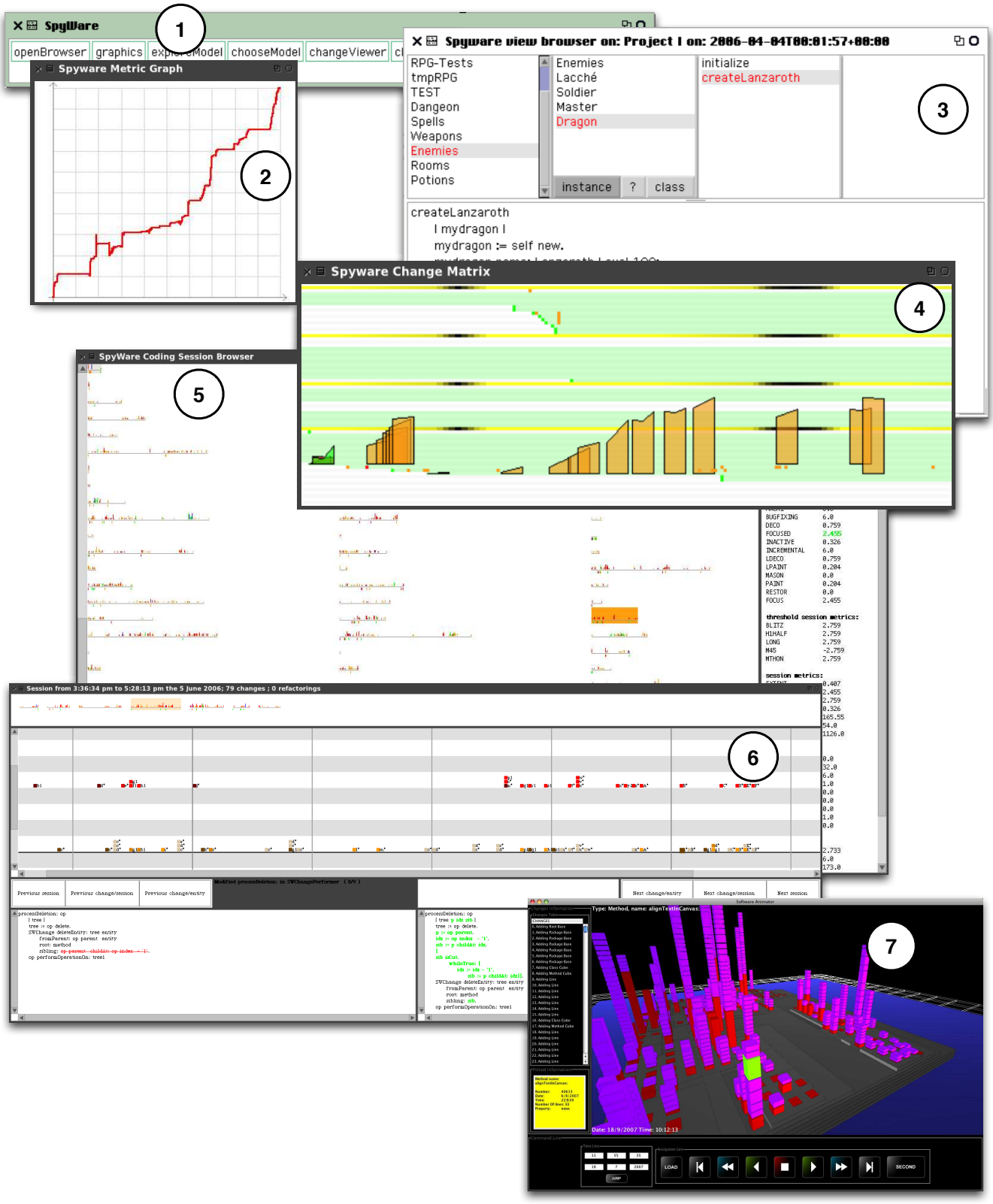

Figure 1: The SpyWare toolset