Towards Change-aware Development Tools

Romain Robbes Michele Lanza

University of Lugano
Faculty of Informatics

Technical Report No. 2007/06
May 2007

Abstract

Software development practice still relies on the notion that programming is equivalent to editing text. This view is also supported by mainstream versioning systems, such as CVS and SubVersion, which are excellent at versioning text files. We argue that `programming = text editing` may have been true years ago, but nowadays we construct complex systems by *changing* them piecemeal. However, despite recent advances which explicitly support change, such as refactorings and agile development methodologies, most development tools in use are ill-suited to deal with software change: they see a system as a collection of text files. This vision leads to code which is harder to understand since its history is not easily accessible, and harder to change since such a representation of code is against the inherently incremental nature of software development.

In this paper we argue that the existing symbiosis between languages and integrated development environments should be extended to support a first-class *change-based* representation of evolving systems. We analyze the shortcomings of current practice, and illustrate our proposal through a scenario describing what such a change-based model of software would accomplish. We then detail our change-aware development model, its ongoing implementation and the promising results we obtained.

1 Introduction

Lehman and Belady's software evolution laws [22] established that successful software systems *must change*, or become progressively less useful in the face of new requirements. Over time, a system becomes harder to maintain because of the increasing complexity induced by the modifications that are necessary to keep the system in use. Architectural erosion [28] and code decay [11] are just symptoms of the diseases affecting such aging systems. This results in higher maintenance costs, estimated between 50% and 90% [12, 23] of the time and money spent over the full system life-cycle[1]. Considering that 10 years ago the estimated yearly expenditure in software maintenance of the United States amounted to more than 70 billion USD [38], a slight amelioration in maintenance would lead to a considerable positive financial impact.

In this "battle against growing costs and complexity" a sadly ignored asset is represented by reverse engineering and reengineering tools, mostly developed by academics, which use a variety of techniques to aid program comprehension (program comprehension activities make up 50% of maintenance [8]). However, the impact of such tools on industrial practice remains still to be seen. Why is that? There is the (outdated) perception that the life-cycle of software is split in a forward engineering and a maintenance phase. Research on software engineering and programming languages is mostly concerned to advance the frontiers of forward engineering, only little effort is spent on pushing program comprehension aids into development environments. This is also due to forward development having a higher esteem, while maintenance tasks are often regarded as second-rate.

Discussing this questionable *status quo* is not the goal of this paper, but it exposes a fundamental flaw in the way people, including developers, perceive software. The very nature of terms used in the context of software development, such as software engineering or software architecture, implies an innate "rigidity" that source code supposedly exhibits: it seems that code should be written to resist change, a paradoxical view in the light of reality.

Despite recent innovations aimed at easing change such as design patterns [18] (proven solutions to recurring problems), refactorings [17] (semi-automated behavior-preserving changes improving program structure) and agile methodologies [5, 6] (acknowledging that change is bound to happen in a system's life and hence preparing for it), the perception of rigid code has not changed much, and the maintenance problems of evolving systems remain the same. Indeed, design patterns make a system more flexible and thus easier to change, but at the price of a higher level of abstraction and an increased complexity. Agile methodologies promote the use of refactorings, but too much refactoring can make developers lose their bearings in a familiar system.

We argue that developers are ill-equipped to deal with continually evolving systems, despite the fact that programming languages and the tools supporting them have greatly improved in the last years, as modern integrated development environments (IDEs) such as Eclipse show. In fact, software engineering research and practice focuses on developing mechanisms to enforce rigorous development and on limiting the effects of change or preventing it altogether: *Change is dangerous*. Most development tools still work under the assumption that a program is a set of editable files: changing them could break the fragile contents. Indeed, programming is regarded as equivalent to text editing or some "kind of writing" [39].

Moreover, the most important tool when maintaining a program, namely the ver-

[1]For more information see http://www.cs.jyu.fi/ koskinen/smcosts.htm

sioning system – the most widely used ones being CVS and SubVersion – enforce this notion by following the checkout/checkin model [16], i.e., a developer checks the most recent version of the system out of the repository, performs some modifications, and once done checks it back into the repository. Versioning systems thus treat program versions as snapshots in time which must be explicitly created by the developers when they commit their changes to the repository. The actual sequence of changes between two snapshots is lost and can only be recovered partially with computationally intensive processes [32].

In short, analyzing software evolution based on the data provided by mainstream versioning systems corresponds to watching a movie where many frames are missing. This leads to wrong assumptions about the nature of software systems and programming in general: programming is an inherently incremental activity in which the code base of a system is continuously changed, not a sequence of well-defined feature additions planned ahead of time: In reality, it is common for unforeseen features to be added to a program.

The research community of software configuration management (SCM) has generated a number of change-based versioning systems, which however compared to what we propose work at a much higher level of granularity [15]: In their case the changes that can be combined to construct versions work at the granularity of system components, i.e., files. Example systems are Aide-de-Camp (now called TRUEChange), PIE[19], and Adele (I + II)[13].

In contrast, we want to explore a novel way of conceiving the phenomenon of evolving software. Our goal is to provide a more accurate model of software evolution by recording the changes *as they happen* in the IDE, and not following the classical checkin/checkout model. In addition, the changes we model have domain specific, object-oriented semantics, whereas versioning systems mainly operate at the file level to be language independent.

Our research is motivated by the conviction that software evolution is the key to productivity [26] in software engineering processes. Consequently we must place change in the center of the software development process and focus our research efforts on activities that enable and facilitate the evolution of complex software systems. These activities belong to both the domains of reverse engineering (understanding changes made to the system) and forward engineering (changing the system).

We claim that development tools must be made change-aware by relying on a model where change itself is acknowledged as the driving force. This implies a shift in the way that programming is perceived, but it still complements the use of current integrated development environments: change-aware tools must be built on top of these IDEs to support a first-class notion of change.

In this paper we describe our approach and its implementation to support our claim. The contributions of this paper are the following:

- A novel change-based approach to model evolving software systems based on recording IDE events. Our approach is loss-less with respect to the information about the evolution of a system, unlike the mainstream versioning systems.

- A description of how change-based evolution can impact development tools in a relevant way by means of tools easing software evolution (forward engineering), and software evolution analysis (reverse engineering).

- An initial validation with the implementation of a number of tools supporting our approach which, exploiting the information retrieved from the IDE, can feed useful information back into the integrated development environment.

Structure of the paper. In Section 2 we depict a scenario in which developers use change-aware tools for their daily chores, and motivate the positive impact that such tools could have. In Section 3 we detail our approach and elaborate on how it helps to tackle the problems of evolving software systems. Section 4 presents our ongoing implementation and the results we obtained. In Section 5 we discuss the advantages and drawbacks of our approach while we discuss related work in Section 6. Finally, Section 7 concludes.

2 Step into the Time Machine

Fast-forward to 2011. Alice and Bob are two developers working on the same project at the multinational FooBar Systems. They are located in different time zones, and Bob starts his working day before Alice. They use both reverse and forward engineering tools in a natural fashion since they are seamlessly integrated in their development environment.

Reviewing and Understanding Recent Changes. Alice starts her day by looking at recent system changes using a *change visualizer*. It displays the recent activity in the system both by developers and by sessions. Alice can see the overall system activity at a glance and infer who is changing what by how much. She notices that Bob has been quite active lately, and she decides to merge his recent contributions with hers. Bob has marked his changes as “ready for integration”.

She launches the *automatic changelog*, which provides an interactive view of the recent changes to the system. She selects Bob’s development sessions from the last week and sees that Bob has performed several refactorings and fixed a certain number of bugs, while adding some functionality to the system. She can examine each change distinctly, and consult the annotations that Bob added using the same tool to explain his bug fixes. She sees that Bob used a refactoring to rename a class. She can ask for details and see the impact of the refactoring in the entire code base if she wishes to.

The functionality Bob added is not trivial, so to understand it, she “steps” through it. Alice is not using a debugger though, she is using a *change replayer*. This tool replays all the actions Bob did in a given period, in the order he did them. It is equivalent to watching a movie of Bob working. Bob annotated his changes with free-form comments explaining the process he employed while implementing the feature. Viewing both how the code was created and the final result allows Alice to better understand the final intent since the code is modified incrementally. The important features are first defined then refined, making it clear what the main point of the changes is.

The tool replays changes one at a time, modifying the code base incrementally. It also uses color-coding to characterize the performed changes: red for bug fixes, blue for refactorings, green for feature addition. This allows Alice to run quickly through the refactorings to focus on the added features. She can also slow down, pause, and even rewind the stream of changes. She quickly identifies the key concepts, whose classes and methods were introduced early during the session. She also identifies some potentially “tricky” code: Bob has modified it heavily before finishing his work. Since it took some time and several attempts for Bob to get the implementation right, the code has some subtleties that make it harder to understand. Hence Alice annotates this part of the system as such, to warn other developers who will encounter this code later.

Locating potential conflicts. Alice decides she understands the new feature well enough to perform the merge, so she opens the *change merger*. It uses change operations at the semantic level, merging entities such as classes and methods. Compared to traditional text-based merging tools, its conflict rate is reduced. The tool informs her that Bob renamed a class, and that it causes a conflict, since she uses it. Would Alice rather cancel the refactoring, or apply it also to her references to the class? She thinks the name Bob used is more descriptive, so she tells the merge tool to adopt the new name in her code too.

Locating Bugs. After the merge process is done, Alice runs the unit tests to make sure the code base is still behaving correctly. There she has a surprise: some of the tests do not pass anymore! Since Bob and her each added several classes, manual debugging will be time-consuming. Thus she opens again the change replayer, telling it to locate the changes responsible for the unexpected behavior. The replayer steps through the change sequence of the code, running the tests at each step. Since the tests are time-consuming, it proceeds by dichotomy rather than sequentially: the first half of the change sequence is considered, and depending on the test outcome, either the first quarter or the first three quarter of the changes are considered for the next step. The algorithm iterates over the changes and after a few moments, Alice locates the offending statements. She then steps back and forth through a change time window to understand the rationale behind the bug-inducing modification. She executes some code at different steps of the history, as she would do in a debugger. She actually uses a debugger, but on several states of the program with several debugger windows open, each on a different version of the system. This *change-aware debugger* allows her to run several versions of the program in parallel, setting breakpoints which are triggered only when two versions of a program have different results at different times. This allows Alice to comprehend the chain of circumstances leading to the bug [42, 43].

Understanding Highly Abstract Code. While reading some of the code she notices that it consists of numerous small methods calling each other. FooBar Systems wants their frameworks to be highly maintainable. They are thus highly abstracted, with as little code duplication as possible. This maximizes reuse, but limits readability since extra degrees of indirection are introduced. The browser she uses is a *concrete browser*: it can optionally show the code in a more concrete way than it actually is. Every method call can be inlined and viewed in place, in the calling method. Relevant type checks are inserted for polymorphic calls, and super calls are treated the same way. She sets up the code browser to automatically inline shorts methods for this browsing session of the program. The browser works by generating change operations inlining code and applying them to the code base at will. Having slightly longer methods improves understandability by augmenting context and limiting the number of entities to understand: Alice no longer has to switch context and browse several methods at the same time, although they still are there.

She quickly discovers the problem: Bob assumed that every instance of class Bar was behaving in a certain manner, and used this assumption in this code. Her modifications changed the behavior of Bar in certain circumstances.

Solving Bugs and Refactoring Code. Alice finds that in the given circumstances performing this behavior change is not trivial. She thinks other variations of the behavior will have to be added in the future. So she decides to break the class Bar into two components. The changing behavior will be encoded in a hierarchy of new classes: AbstractBaz, BobBaz and AliceBaz. Bar will delegate the variable behavior to one of these classes at runtime, through an instance variable. She will find better names later, as renaming is instantaneous: in her environment, the name of a class is just a property, independent from its identity.

She starts the process by creating a new instance variable baz in the class Bar containing the new class, and delegates methods implementing the variable behavior to baz. She transfers the body of the method to one of the subclasses of AbstractBaz. Some instance variables of Bar must also be replaced by accessors in the process, making it a bit tedious. Alice knows that this can be easily automated: She goes back to her *change history*, select a sequence of operations representing the changes she just made (the delegation of the method to baz, including replacing references to variables by accessors), and attempts to generalize it. This opens a *change editor* on the sequence of changes. In it, she can decide which part of the change is generic, and which part is concrete. She defines a generic change operating on two classes A and B, an instance variable v and a set of methods S. This change moves the implementation of the methods in S from A to new methods in B (creating accessors for variables still belonging to A if necessary) and replaces the bodies of the methods in S with delegation stubs that forward the messages to v, which is an instance of B. She then instantiates the generic change: She fixes the target class (Bar), the destination class (BobBaz), selects the set of methods corresponding to the variable behavior, and tells the *change editor* to delegate the behavior to instance variable baz. Finally she applies the generalized change, which processes the sequence of methods and modifies the code base. She switches to her branch of the code, and applies the generalized change again, this time using AliceBaz as a parameter instead of BobBaz. After merging again, the tests run green.

Collaborative Work and Awareness. At this point, Bob, who is still working, is notified by the *background change merger* that some recent changes he made can not be merged with Alice's version. Changes made by developers are broadcast to other participants. When the IDE receives a change, it attempts to merge it with the local changes in a parallel model of the code, without impacting the code the developer is working on. Only if the merge fails is a notification issued to both developers, so that they can contact each other. Once Bob sees Alice has stopped making changes (indicating she is less busy), Bob enters in contact with her to see what the problem is. They quickly figure the problem out, and Bob adapts his code to the latest modifications early on. Furthermore, while discussing with Alice, they found some better names than the clunky AliceBaz and BobBaz.

Maintaining Duplicated Code. Bob has to adapt his code to Alice's changes. While modifying a method, the environment asks him via the *clone tracker* if he wants to apply the same change to possible clones of the method. Bob reviews each instance the environment presents him and applies all or part of the changes to those methods who needs it. Bob knew he had to look up for duplicated code, but forgot exactly which methods needed to be changed. Since several instances were real clones, he marks these as refactoring candidates. Later on, he will refactor them to comply to FooBar Systems' high standards for maintainability.

2.1 Change-Aware Tools: Wishful thinking?

The scenario above contains an informal description of several fictive change-aware tools. In this section we describe in detail what such tools could help to achieve, hint at our own tool implementations and compare them with already existing tools having similar functionality, if any exists.

The System Change Visualizer acts as an entry point to the system when reviewing the changes which have been applied to it. It helps to give a detailed answer to the question "What exactly happened since I last touched the system?". Its goal is to guide the user to interesting parts of the system through advanced visualizations and metrics of the system and its recent history. Several tools already exist to monitor the evolution of a software system both using metrics and visualizations, such as the free Maven[2]. However the tool set we implemented and for which we provide an in-depth description in Section 4, monitors changes at a much finer-grained level and can display changes from the package level up to the statement level. Furthermore it tracks changes *as they happen* in order to give a detailed trace of what has been done during a development session. Approaches extracting knowledge from a source-code repository are limited by the checkin/checkout they follow and therefore cannot display such data as it is not captured by the versioning system [30].

The Automatic Change Log provides a textual description generated from the changes done during a selected period. It can display higher-level change information such as refactorings, and the lower-level changes constituting them. Individual or groups of changes can be annotated by programmers. It hence explains a series of changes in more detail than the *System change visualizer*, without going into implementation details. Tools like this exist, such as the automatic changelog provided by SubVersion [3], but the envisioned tool could exploit the composite nature of changes to allow the same kind of "drill-down" approach offered by the system change visualizer. Being integrated with an IDE, such a tool would be interactive, rather than a static text file generator.

The Change Replayer takes a sequence of changes and interprets them as actual developer actions. With it the actions a programmer took while using the IDE can be replayed, enabling the understanding of a development session by looking at the changes in the order in which they actually happened. It shows concepts being introduced in a development session in their most basic form, and then extended to handle special cases and extra enhancements secondary to the overall comprehension of the new feature. To our knowledge, no current tool records enough information to allow this.

The Change Merger uses fine-grained changes to merge concurrent modifications to a program. Versioning systems address the merging problem in various ways [25]. Several tools or versioning systems are refactoring-aware [20, 9] and can apply previously performed refactorings to code bases in order to assist in program evolution. Operation-based merging [24] merges sequences of change operations instead of program states.

[2]See http://maven.apache.org/
[3]See http://www.core.com.pl/svn2log

A change-aware versioning system would do the same but at a finer-grained level, also relying on the same common change repository available to other tools.

The Change-aware Debugger leverages the history of a software system as an aid to debugging. A few approaches have considered the use of evolutionary information to aid debugging, such as delta debugging [41] and relative debugging [1]. A debugger following the same principles but based on a change-based representation of data would allow to precisely locate an error in time, but also to help finding the cause of a bug since the changes before and after the bug-inducing change provide a richer context to understand why such a change was introduced in the system.

The Concrete browser enables the user to see the implementation of a method in the context of its usage. The inline method refactoring does the same, but it is manually triggered to improve the structure of the code base and is expected to have a permanent effect [17]. In contrast the tool we describe would temporarily inline method calls and super calls to expand the working context of a programmer. Inlining is also used for performance optimisation, but to our knowledge it has not been tried yet to facilitate program comprehension.

The Change Historian allows one to access the change history of a specific entity or the entire system.

The Change editor/generalizer uses the change history to fetch a sequence of recent changes and then provides an interface to convert this concrete change into a general system transformation. Tools such as the Rewrite tool [34], the Design Maintenance Systems [2], and the general field of Model Driven Engineering[4], provide the possibility to transform models of programs. However they are mostly intended for large-scale transformations: they rely on special syntaxes to write transformations and have a steep learning curve. Refactorings tools [35] integrated in IDEs are widely used in practice but automate a small number of operations. The change editor (or change generalizer) is designed to be used in everyday tasks, filling the gap between complex transformation tools and semi-automated refactoring tools. It relies on the ability to query the recent history of a system to retrieve one or more concrete instances of a change, and then provides an interface to generalize the change and apply it to other parts of the system. This approach would allow one to avoid the steep learning curve of a dedicated transformation language.

The Background Merger informs developers of conflicts in collaborative work. Conflicts can hence be resolved early, before they become too problematic. Tools such as Palantìr [36], TUKAN [37], and to some extent Céline [14] support awareness of developer activity. A change-based awareness tool would provide the same functionality, with additional support for the different kind of changes provided by the change model (such as refactorings). It would thus provide more information about the conflict-inducing changes, which could in turn affect how developer would use it. A conflict provoked by a refactoring might not be of the same importance than a conflict caused by a bug fix.

[4]See http://www.omg.org/mda/

The Clone Tracker tracks closely related code fragments and offer to apply changes to all clones when one is modified. Several clone detection tools exists [10], [40], [3], [4], but little has been done in automatically processing these clones to help their maintainability: only empirical studies have been done on the clone data, such as [21]. In addition, we think an IDE equipped with monitoring tools is a valuable help to detect occurrences of clones: the search for clones can be focused by looking deeper into the edition of entities close in time, or the use of copy and paste operations. A clone tracking tool based on a change-aware IDE could hence be more accurate than other clone detection tools, and also use the same mechanism outlined in the change editor to propagate changes done to one instance of a clone to the other instances present in the system.

2.2 Back to Reality

Some of the functionality we describe has already been implemented in stand-alone tools. However we claim that such tools would benefit both from an integration based on a common change-based representation of software and from the interoperability offered by them being built on top of an IDE. One of the strengths of IDEs lies in their close integration of tools, which allows information to be exchanged between them. For example, IDEs allow a compilation error to be directly noticed in the editor, which eases its correction. In the same fashion, change-aware tools collaborating together will offer a benefit greater than their individual advantages. Furthermore, a change-based model of software still offers some unique advantages over these tools, that we focus on in the next section.

To sum up, we expect a change-aware development environment supporting these fictive tools to have a significant impact on systems built with it and on developers using it. Systems would be more amenable to change with the presence of advanced refactoring tools and easy-to-use program transformations. Code clones would be less of a maintenance problem as they could be tracked and evolve together if they cannot be refactored in a more abstract design. Collaborative work would be made easier by the presence of fully integrated awareness tools and code merging capabilities based on the same change model as the other tools.

In the context of program comprehension and maintenance, a change-based IDE also provides assistance since it records the history of the system in its most finest details. The history can be replayed at will with an extended context for each change, and can play a valuable role in debugging. Moreover, if a system changes faster due to better tools (such as refactorings and the tools we described), it needs better change tracking facilities so that developers do not lose their tracks in a rapidly changing system.

After this excursion into fiction, let us turn back to reality in the next section, where we present a model of change-aware software development and our implementation.

3 Change-aware Software Development

Our approach to provide support to change-aware development is based on the following principles:

1. *Using the IDE as information source.* The first step is to record an accurate history of the evolution of systems. Thus we built an IDE plugin (described in

detail in a latter section) which records *all* changes effectuated by a programmer to the system.

2. *Modeling change as a first-class entity.* We developed a model of software evolution based on first-class change operations to better match the incremental process of developing software.

3.1 Using the IDE as Information Source

Programmers use IDEs for their day-to-day tasks, since they are powerful tools featuring code completion, refactoring support, incremental compilation, unit testing, advanced debugging, source control integration, quick browsing of the system, etc. A number of them, e.g., Eclipse, are also extensible by plug-in systems.

IDEs are able to provide the functionalities beyond simple text editing because they have an internal reified model of the software system being built. We advocate using the IDE itself as the source of evolution information instead of a versioning system – as done by traditional evolution analysis approaches – because it provides the best means to get as close as possible to the developer's intentions.

Most IDEs feature an event notification system, so tools can react to what the developer is doing. IDE hooks monitor when a change happens, such as a class compilation or a method refactoring. The approach we propose uses these IDE hooks to react when a developer modifies the system by creating *first-class change entities* to mirror his actions.

3.2 Modeling Change as a First-class Entity

First-class change entities are objects modeling the history of a system following the incremental way it was built. They contain enough information to reproduce the program of which they model the history. When executed, they yield an abstract representation of the program they belong to at a certain point in time. They also contain additional information useful for evolution researchers, such as who performed which change operations, and when it was carried out.

Change operations also model with greater accuracy the way the developer thinks about the system. If a developer wants to rename a variable, he does not think about replacing all methods referencing it with new methods, even if that is what the IDE ends up doing: Modeling incremental modifications to the system eases its understanding.

3.2.1 Program Representation

We model software evolution as a sequence of changes that take a system from one state to the next by means of semantic transformations [32, 31]. By semantic transformation we intend changes at a level high enough to have object-oriented semantics. These modifications are for example the modification of the body of a method and its subsequent compilation, but also include 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* that brought the system to its actual state.

Our approach represents programs as domain-specific entities – at the programming language level – rather than text files. We focus on object-oriented programs, and therefore store and analyse constructs such as classes and methods, not files and lines.

We represent a software system as an evolving abstract syntax tree (AST) containing nodes which represent packages, classes, methods, variables and statements. This AST represents one state the program went through during its evolution. Each AST entity has a *change history* containing all change operations applied to it during the system's evolution. A node *a* is a child of a node *b* if *a* contains *b* (a superclass is not the parent of a subclass, only packages are parents of classes). Nodes have *properties*, which vary depending on the node type, such as: for classes, name and superclass; for methods, name, return type and access modifier (public, protected or private, if the language supports them); for variables name, type and access modifier, *etc.*

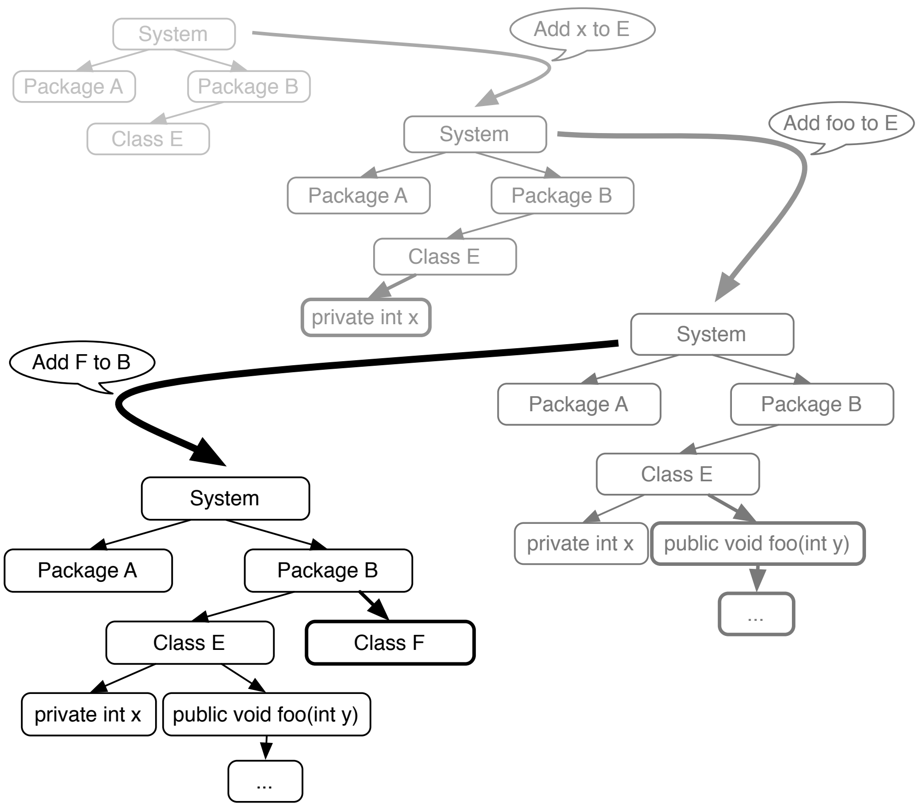


Figure 1: Three changes executing on an AST

3.2.2 Change Operations

Change operations represent the evolution of the system under study: They are actions a programmer performs when he changes a program, which in our model are captured and reified. They represent the transitions from one state of the evolving system to the next (see Figure 1). Examples of change operations are: adding/removing class/methods to/from the system, changing the implementation of a method, or refactorings. The change metamodel is shown in Figure 2. We support *atomic* and *composite* change operations.

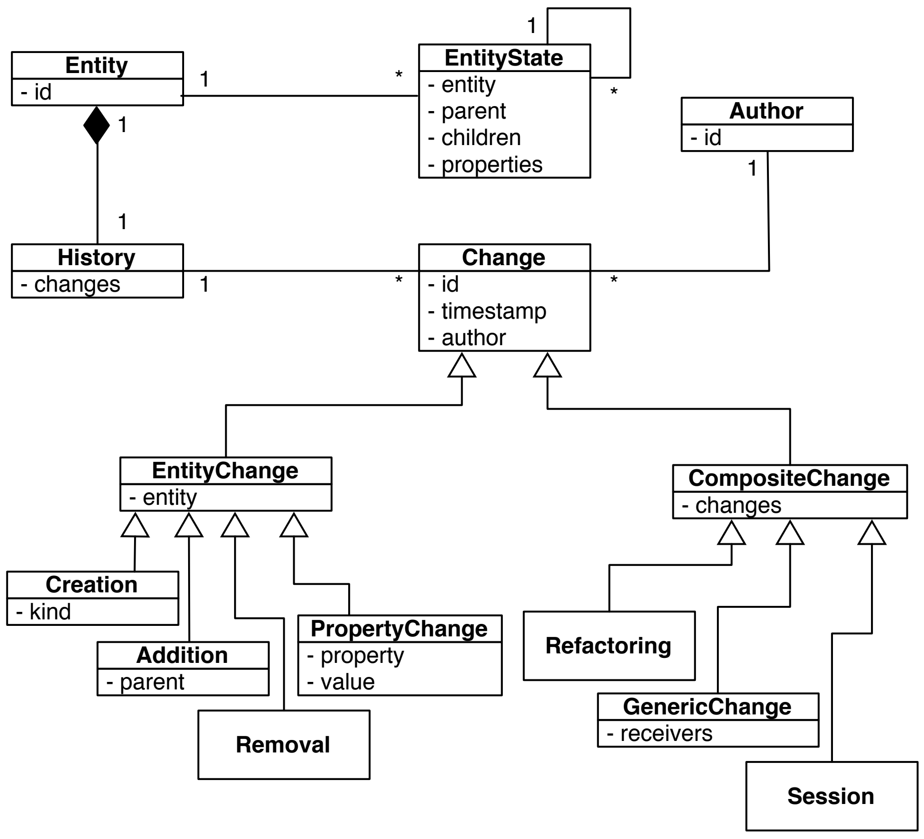


Figure 2: The change metamodel

Atomic Change Operations. Since we represent programs as abstract syntax trees, atomic change operations are, at the finest level, operations on the program's AST. Atomic change operations are executable. By iterating on the list of changes we can generate all the states the program went through during its evolution. The following operations suffice to completely model the evolution of a program AST:

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 children of its parent *p*.

Property change: changes value *v* of property *p* of node *n*.

Composite Change Operations. While atomic change operations are enough to 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: An abstraction mechanism is necessary. Hence change operations are composable, and can be abstracted into higher-level composite change operations. For example, moving a class from one package to another consists in removing it from a package and adding it to another package. These two atomic change operations can be grouped in a single *move class* change operation. There are several levels of increasingly coarser-grained change operations:

Developer-level action: A unit of change from the developer's viewpoint. For example, changing the definition of a class or adding a method are developer-level actions. A developer-level action can contain several atomic changes: A method addition contains changes related to the creation and the addition of its body statements.

Refactoring: Refactorings are behavior-preserving automatic code transformations [17]. For example, the *rename method* refactoring involves changing a method's name, and all references from the old method name to the new name. These developer-level actions can be grouped into a higher-level entity representing the refactoring itself. In the same fashion, the *extract method* refactoring replaces a section of code from a method with a call to a newly created method containing this code fragment. These two changes can also be grouped to form a composite change operation.

Bug fix: Consists of all changes necessary to fix a given bug.

Development session: It aggregates all the changes done during a single development session by a developer, be they bug fixes, refactorings or developer actions. This is the closest in terms of size with a commit extracted from a state-of-the-art versioning system.

3.3 Supporting forward engineering

Our approach as described so far is only able to record the evolution of a system, but not to affect it as required by some of the tools we describe. Tools such as the change editor or the change merger need to actually change a program. To do so we need to extend our model. We support program transformations with the following characteristics:

- Our change objects contain enough information to be executed or undone by themselves. Taking a sequence of change operations and executing them on a model A effectively creates an altered model B of the system. The opposite process is performed by reversing the change sequence and then undoing it.
- Changes can be generated rather than simply recorded. From the model's point of view, it does not matter where the changes come from. They can hence be generated by tools such as the change editor or the clone tracker described previously.
- General transformation are handled in our model by the concept of a generic change (see Figure 3), which is effectively a template which can be parametrized to generate concrete instances of the change it represents. We believe such a mechanism is sufficient to model the transformations we have in mind, but have yet to confirm it.

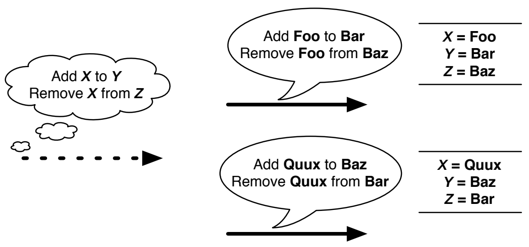


Figure 3: A generic change instantiated in 2 concrete cases

3.4 Summary

To sum up, our approach consists of the following steps: (1) Using the hooks of the IDE to be notified of developer activity. (2) Reacting to this activity by creating first-class change objects representing the semantic actions the developer is performing. Change objects can also be created by other tools to model transformations to the program. (3) Executing these change objects to move the program representation to any point in time, or to transform the program if changes are generated by change-aware tools rather than the developers.

4 From Fiction to Reality: SpyWare

We have an ongoing implementation of a change-based environment in the form of SpyWare[5], a platform built to support change-aware extensions to the Squeak Smalltalk

[5]A web-based demo of some of SpyWare's functionality is available at http://romain.robb.es/spyware.

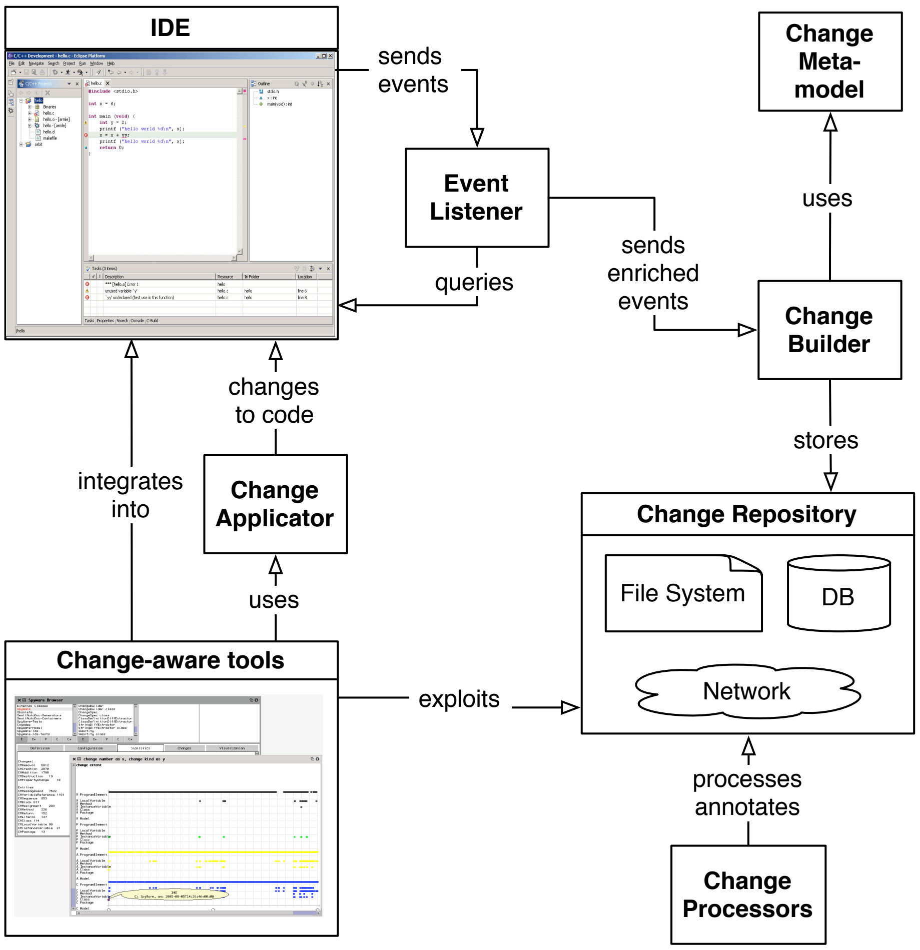


Figure 4: SpyWare's architecture.

IDE.

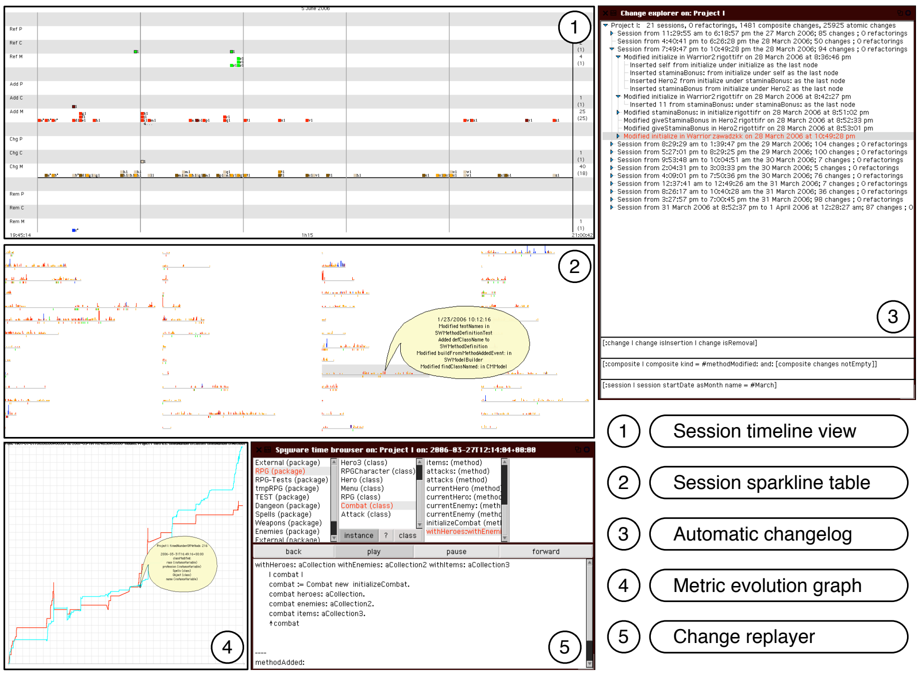
We first present SpyWare's architecture, then describe the tools for which we have a prototype implementation. We then demonstrate the benefits of IDE integration and finally validate the implemented tools' usefulness by recalling results of previous case studies.

4.1 SpyWare's Architecture

SpyWare is constituted of the following components (see Figure 4):

Change Metamodel. All the classes necessary to instantiate our change meta-model for a given language and IDE are implemented in this component.

Event Listener. This module is registered in the Integrated Development Environment's notification system to gather the information needed by the change builder. It collects event objects issued by the IDE, and queries the IDE for additional information it needs, such as time and authorship information.



```
Project I:   21 sessions, 0 refactorings, 1481 composite changes, 25925 atomic changes
  Session from 11:29:55 am to 6:18:57 pm the 27 March 2006; 85 changes ; 0 refactorings
  Session from 4:40:41 pm to 6:26:28 pm the 28 March 2006; 50 changes ; 0 refactorings
  Session from 7:49:47 pm to 10:49:28 pm the 28 March 2006; 94 changes ; 0 refactorings
    Modified initialize in Warrior2 rigottifr on 28 March 2006 at 8:36:46 pm
      Inserted self from initialize under initialize as the last node
      Inserted staminaBonus: from initialize under self as the last node
      Inserted Hero2 from initialize under staminaBonus: as the last node
      Inserted staminaBonus from initialize under Hero2 as the last node
    Modified initialize in Warrior2 rigottifr on 28 March 2006 at 8:42:27 pm
      Inserted 11 from staminaBonus: under staminaBonus: as the last node
    Modified staminaBonus: in initialize rigottifr on 28 March 2006 at 8:51:02 pm
    Modified giveStaminaBonus in Hero2 rigottifr on 28 March 2006 at 8:52:33 pm
    Modified giveStaminaBonus in Hero2 rigottifr on 28 March 2006 at 8:53:01 pm
    Modified initialize in Warrior zawadzkk on 28 March 2006 at 10:49:28 pm
  Session from 8:29:29 am to 1:39:47 pm the 29 March 2006; 104 changes ; 0 refactorings
  Session from 5:27:01 pm to 8:29:25 pm the 29 March 2006; 100 changes ; 0 refactorings
  Session from 9:53:48 am to 10:04:51 am the 30 March 2006; 7 changes ; 0 refactorings
  Session from 2:04:31 pm to 3:03:33 pm the 30 March 2006; 5 changes ; 0 refactorings
  Session from 4:09:01 pm to 7:50:36 pm the 30 March 2006; 76 changes ; 0 refactorings
  Session from 12:37:41 am to 12:49:26 am the 31 March 2006; 7 changes ; 0 refactorings
  Session from 8:26:17 am to 10:40:28 am the 31 March 2006; 36 changes ; 0 refactorings
  Session from 3:27:57 pm to 7:00:45 pm the 31 March 2006; 98 changes ; 0 refactorings
  Session from 31 March 2006 at 8:52:37 pm to 1 April 2006 at 12:28:27 am; 87 changes ; 0

[:change | change isInsertion | change isRemoval]
[:composite | composite kind = #methodModified: and: [composite changes notEmpty]]
[:session | session startDate asMonth name = #March]
```

```
External (package)      Hero3 (class)           items: (method)
RPG (package)           RPGCharacter (class)    attacks: (method)
RPG-Tests (package)     Hero (class)            attacks (method)
tmpRPG (package)        Menu (class)            currentHero (method)
TEST (package)          RPG (class)             currentHero: (method
Dangeon (package)       Combat (class)          currentEnemy: (meth
Spells (package)        Attack (class)          currentEnemy (metho
Weapons (package)                               initializeCombat (met
Enemies (package)                               withHeroes:withEnemi
External (package)
```

```
withHeroes: aCollection withEnemies: aCollection2 withItems: aCollection3
    | combat |
    combat := Combat new  initializeCombat.
    combat heroes: aCollection.
    combat enemies: aCollection2.
    combat items: aCollection3.
    ↑combat


----
methodAdded:
```

Figure 5: Several tools built on top of SpyWare

Change builder. This component translates event objects to a sequence of changes. Since the change builder is notified of events when a method is compiled and we wish to track changes down to the statement level, we require a code difference engine relying on a code parser of the language we monitor. Contrary to other approaches, we only need to analyse one method at a time, and not two possibly large programs at once. As a consequence we can easily trade some computing time for extra accuracy in our analysis.

Change applicator. Several tools need to display or execute the program at any given state. To achieve this we need to reproduce any state of the model and translate it back in the original programming language. This component deals with the intricacies of this task.

Change repository. This component deals with the persistency of the model. Since we rely on first-class change operations, these are the only entities we need to store. So far we use a simple approach based on text and binary files. In the future we plan to use a database. Later on this component will also broadcast changes to the environments of other developers.

Change-aware tools. Based on the other components, these tools form the user-visible part of our prototype, allowing the user to interact with the model.

4.2 Evolution Analysis Tools

The aspect we tackled first is reverse engineering and software evolution analysis. Program analysis tools only need to operate on a model of the software which is "inert", as they do not modify the software itself: they just analyse its past history and current state to assist its comprehension. Working on this kind of tool allowed us to define a first iteration of our change metamodel. We then built prototype versions of some of the fictive tools we described, shown in Figure 5, Figure 6, and Figure 7: the system change visualizer, the automatic changelog, and the change replayer.

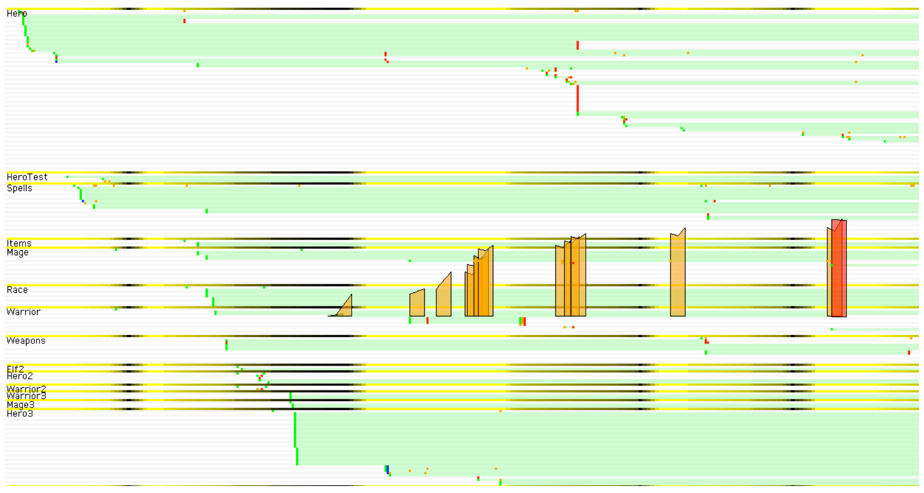
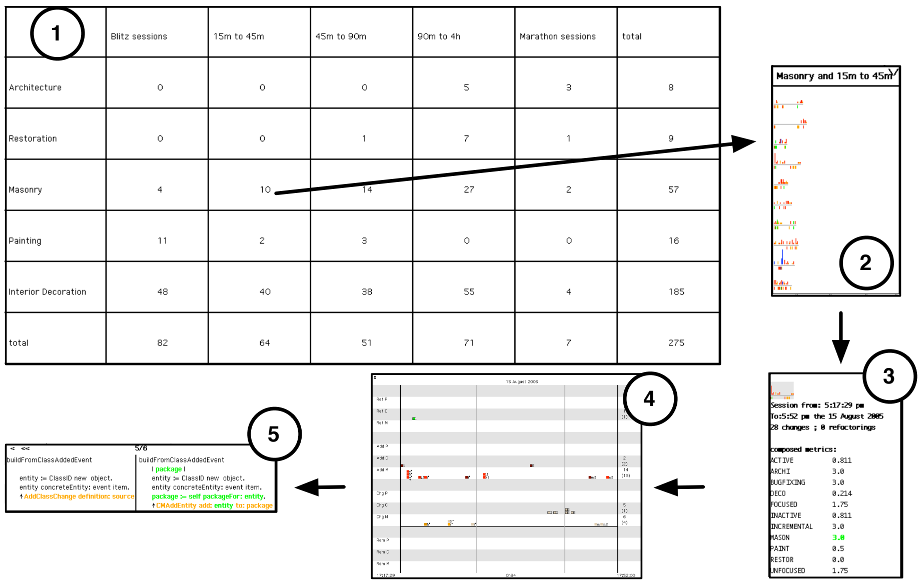


Figure 6: The change matrix of one student project, showing the details of one method

The system change visualizer. It provides a set of visualizations and metrics focused on displaying the system under study as a set of sessions and allowing the user to quickly select sessions of interest. It features:

- System-level metrics, such as number of classes, number of methods, average size of methods. These metrics are defined at the programming language level.
- An interactive global system view named the change matrix (Figure 6), in which the evolution of packages, class and methods can be assessed at a glance.
- Session-level metrics such as session activity, or the average number of changes per entity (telling if a session if focused or not) are based on the change data and are used to classify and characterize sessions.
- Session sparklines display the time-based activity of a session in a compact way.
- Session inspectors provide details on a specific session.
- Session timeline views display the activity in a session down to the level of individual changes.

The automatic changelog. It is a tree-based widget displaying the entire history of a system as its sessions. Sessions can be expanded to show the changes occurring during them, including changes at a higher-level such as refactorings. These changes can be also expanded to show lower-level changes at the AST level. It features a simple query interface with which the scope of the tool can be reduced.



	Blitz sessions	15m to 45m	45m to 90m	90m to 4h	Marathon sessions	total
Architecture	0	0	0	5	3	8
Restoration	0	0	1	7	1	9
Masonry	4	10	14	27	2	57
Painting	11	2	3	0	0	16
Interior Decoration	48	40	38	55	4	185
total	82	64	51	71	7	275

```
Session from: 5:17:29 pm
To:5:52 pm the 15 August 2005
28 changes ; 0 refactorings

composed metrics:
ACTIVE         0.811
ARCHI          3.0
BUGFIXING      3.0
DECO           0.214
FOCUSED        1.75
INACTIVE       0.811
INCREMENTAL    3.0
MASON          3.0
PAINT          0.5
RESTOR         0.0
UNFOCUSED      1.75
```

```
< <<                          5/6
buildFromClassAddedEvent

    entity := ClassID new object.
    entity concreteEntity: event item.
    ↑AddClassChange definition: source
```

```
buildFromClassAddedEvent
    | package |
    entity := ClassID new object.
    entity concreteEntity: event item.
    package := self packageFor: entity.
    ↑CMAddEntity add: entity to: package
```

Figure 7: Selecting sessions for fine-grained session understanding

The change replayer. It is a source code browser equipped with additional time-controlling buttons to make the browser step back or forward in time. It also focuses on the entity affected by the current change, and shows the changes happening to it using syntax highlighting of source-code elements. It can be set up to display changes at the scale of the entire model, or restrict its view to a single program entity.

4.3 Effects of IDE integration

These tools, metrics and visualizations are interactive and take advantage of the integration with the IDE to reference each other extensively. Selecting sessions for further inspection and fine-grained understanding follows an interactive process, exemplified in Figure 7:

1. The user opens a session table view, in which sessions are classified according to one or two sets of session-based metrics, such as length of the sessions, the activity level, or the kind of changes performed in the session [33].

2. Each cell of the table can be clicked on to display the sessions fulfilling these criteria – such as "long sessions with a high activity" – as sparklines, a compact representation of the sessions with a high density of information: it shows how the development went during the session from a high-level view.

3. Sparklines, when clicked, open a session inspector, displaying all the metrics available for a given session.

4. The inspector can summon a session timeline visualizer which displays a timeline view of the session in which individual changes are displayed

5. Each change, when clicked, brings a compact version of the change replayer focused on it.

In the same fashion, the change matrix in Figure 6 displays all the changes in the system (or a part of it) in a succinct way. It can display on demand more information about a given class or method by providing details about each change performed to the class/method. It is also integrated with the change replayer in the same way the session timeline visualizer is.

Most elements of visualizations can display tooltips as a preview of their contents to help the user decide whether he should click on them or not. Figure 5 shows a session sparkline and a metric graph displaying tooltips.

4.4 Validation of evolution analysis tools

Our evolution analysis tools have been validated on several case studies:

- The metrics graphs – displaying the evolution of one or more system-level metrics over time – have been validated on 9 projects implemented by students at our university [32]. The metrics graphs were used to infer high-level characteristics of the projects, and also to visually identify sessions of activity of students. The shape of the metric curves was also used to do a primary characterization of the development sessions: if the system size follows an irregular trend during a session, it is an indicator that the developer was unsure of himself or looking for a defect by trial and error.

- The change matrix [31] was then used to gain some deeper insight on the same student projects. Sessions can be visually identified, and medium to high-level decisions about the design can be inferred from the order in which classes were changed, as well as the extent of modifications applied to them and their methods. A rather accurate picture of the development plan of a small-scale project (or a particular feature of a larger project), can be gained by analysing this matrix.

- The session-based characterization was used with two larger case studies, SpyWare itself and another project we monitored for a few months, to select development sessions and understanding those in details, according to the process outlined in Section 4.3, and in Figure 7.

We believe these results show that fine-grained evolutionary information is useful to understand a system at a finer level than conventional evolution analysis tools. Our tools can also provide a higher-level view of the system's evolution if needed. These results are in accord with the scenario we outlined, in which developers both need to understand how the system changed and how individual entities changed when they need to focus in a particular task.

5 Reflections on Change-Based Evolution

Advantages. Modelling the evolution of software systems as a change-based process and gathering evolutionary data has several advantages over a version-based view of evolution extracted from a versioning system:

- *Incrementality.* Reacting to events as they happen gives us more accurate information than mainstream versioning systems. Timestamps are more precise, and not reduced to commit times. Events happen one by one, giving us more context to process them than if we had to process a batch of them, originated from an entire day's work.

- *High level.* It is significantly easier to maintain a semantic representation of the model of an evolving software when using an IDE. Events originating in the IDE are high level. Their granularity is the one of classes and methods, not files and lines. Code parsing is required only at the method level.

- *Closer to reality.* Representing the evolution as a sequence of changes actually matches the way developers implement and think about software, and is hence a closer fit to the process of software evolution.

- *Granularity.* Every program entity can be modeled and tracked along its versions, down to the statement level if necessary. There is no state duplication, leading to space economization when an entity does not change during several versions.

- *Increased software flexibility.* Going back and forward in time using change objects is easy. It leads to more experiments with the code base, easing "trial and error" in development. Transformations of the code are easier to implement when the entire history of a system is constituted of transformations.

Drawbacks. We have identified a number of possible issues and negative implications to our approach:

- *Absence of IDE.* There are still people that do not use IDEs to develop software, but only plain text editors and command-line compilation. To make it short, our approach is useless and not applicable in such a context.

- *Significant porting effort.* Our approach is language-specific, and to some extent IDE-specific. Hence porting our approach to a new language or Integrated Development Environment involves a significant effort since the model has to be fine-tuned to the language under study and the tools have to be ported to the new IDE to benefit maximally from it. However, a first step implementing only the event listener and the queries it performs to the IDE is relatively simple. It allows several evolution analysis tools to function since they can operate – albeit with reduced functionality – outside of the IDE. This has been done for the Visualworks IDE, and is an ongoing effort for Eclipse.

- *Validation through case studies.* Since the data we rely on was not previously captured we cannot use already existing applications as a data source for evolution analysis tools: such applications will have a truncated history. To address this we monitored our prototype itself as a first case study. At this time of writing, SpyWare includes around 300 classes, making it a medium-sized case study. We have also collected data from student projects and have another case study from an independent developer, for a total of 11 case studies. Furthermore, several other case studies have started, and we will actively promote our prototype in open-source communities.

- *Performance.* We are not focusing on performance at the moment. We do not know if our approach can scale to very large applications, *i.e.,* if representing millions of lines of codes as a sequence of incremental changes would be too resource-intensive. Caching, and making periodic checkpoints in the system to transfer some change objects out of the main memory, are possible optimization techniques. At the moment, performance is still acceptable and therefore not an issue.

- *Acceptance.* People do not like changing their habits. For example, CVS is still heavily used despite its flaws, and Subversion is only gathering momentum because it is close enough to CVS. In a forward engineering context it will take some time until people get used to the notion of first-class change.

Relation to the SCM Field. Configuration management and versioning systems are already recording the history of software systems. One could think our approach is equivalent and as such amounts to building another versioning system. We do not attempt to do so for several reasons:

- The SCM community has different assumptions about a software system than we have. The philosophy of versioning systems is to be language and application independent. In contrast our approach is language-specific to gather as much information as possible on evolving systems. Researchers in SCM see the field of software evolution analysis as a side effect of theirs [15] – which is true so far –, whereas our aim is to explicitly support it. However, Estublier and other well-known researchers from the SCM field argue that breaking the assumption of language-independence is the next stepping stone in SCM [15], suggesting that the language-independent field of SCM is extensively covered.
- An SCM system does much more than just recording the history of an evolving system. It has to handle conflicts between version, provide configuration selection, handle branches, manage the build process of a system, *etc.* These features are not our focus. Although we described tools easing merging of software, we do not intent to provide a full SCM solution in the near future.
- One of the main drawbacks of our approach is the lack of case studies. Since developers do not want to change their habits, they tend to be loyal to their SCM system and will not change easily. Relying on an SCM solution would only raise the barrier to entry and make our tools harder to accept.
- We advocate the need to make tools *change-aware*, while advanced SCM systems[7] achieve tool integration by relying on virtual file systems or workspaces to *hide* their presence from other development tools.

6 Related Work

Section 2 already covers tools which have functionality close to the change-aware tools we described. This section covers more general work which bears a resemblance to ours.

Nierstrasz *et al.* advocate that change should be supported at the language level and outlines several mechanisms to address the problem of software evolution [27]. Among them are ChangeBoxes [44], which also introduces an explicit model of software change. However, both models differ significantly since the two approaches tackle the problem from different angles. ChangeBoxes primarily aim at supporting change at runtime, allowing several program version to coexist at the same time. Our model is aimed at supporting software evolution in a more static way, but is more complete: it models change up to the statement level while Changeboxes stop at the method level. We think both models could complement each other.

Pluquet and Wuyts [29] present an approach in which every object in a system can be versioned and show several applications of this technique. Their approach is more general as it applies to every object in a system, but it relies on classical state-based versioning of objects, not an explicit change-based representation.

The literature shows that a few versioning systems have attempted a change-based, rather than version-based, approach to versioning [24] [25]. However, these approaches are solely focused on merging, whereas our model allows the change operations to be used in various ways.

7 Conclusions

In this paper we argued that a better support for program evolution is critical in the face of the inevitable changes which need to be performed on existing systems.

We introduced a novel approach to support software evolution in which change itself is acknowledged by being explicitly modelled in integrated development environment. The change-based evolution of systems being built by developers is recorded when they use the IDE and stored in a change-based repository.

This change-based repository can then be used to assess the system's evolution with reverse engineering tools. Advanced development tools can also take advantage of the repository and the change-based model of software evolution to ease the future evolution of systems. We outlined these tools and their features in a scenario, and implemented some of them and have initial results to support our claims.

Much work remains to be done before a truly change-aware development environment sees the light. Our implementation is still at the prototype status and currently only supports evolution analysis tools. Our future work lies in implementing some of the tools we described to support forward engineering, as well as continuing to validate our reverse engineering approach by gathering more case studies. To get out of the prototype status, several aspects such as multi-user support need to be taken into account. We also need to port our tools to a widely-used IDE such a Eclipse.

In short, many cha(lle)nges are awaiting us.

References

[1] D. Abramson and R. Sosic. A debugging and testing tool for supporting software evolution. *Autom. Softw. Eng.*, 3(3/4):369–390, 1996.

[2] R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis, and K. R. Luecke. Reengineering c++ component models via automatic program transformation. In *WCRE*, pages 13–22. IEEE Computer Society, 2005.

[3] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[4] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[5] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[6] A. Cockburn. *Agile Software Development*. Addison Wesley, 2002.

[7] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[8] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[9] D. Dig, T. N. Nguyen, K. Manzoor, and R. Johnson. Molhadoref: a refactoring-aware software configuration management tool. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 732–733, New York, NY, USA, 2006. ACM Press.

[10] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 109–118. IEEE Computer Society, Sept. 1999.

[11] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[12] L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Pro*, pages 17–23, May 2000.

[13] J. Estublier and R. Casallas. The adele configuration manager. In W. F. Tichy, editor, *Trends in Software: Configuration Management*, volume 2, pages 99–134. Wiley, 1994.

[14] J. Estublier and S. Garcia. Concurrent engineering support in software engineering. In *ASE*, pages 209–220. IEEE Computer Society, 2006.

[15] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4):383–430, Oct. 2005.

[16] P. H. Feiler. Configuration management models in commercial environments. Technical report cmu/sei-91-tr-7, Carnegie-Mellon University, Mar. 1991.

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

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[19] I. P. Goldstein and D. G. Bobrow. A layered approach to software design. Technical Report CSL-80-5, Xerox PARC, Dec. 1980.

[20] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings International Conference on Software Engineering (ICSE 2005)*, pages 274–283, 2005.

[21] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proceedings of European Software Engineering Conference (ESEC/FSE 2005)*, pages 187–196, New York NY, 2005. ACM Press.

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

[23] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of applications software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[24] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, New York, NY, USA, 1992. ACM Press.

[25] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[26] O. Nierstrasz. Software evolution as the key to productivity. In *Proceedings Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, Oct. 2002. preprint.

[27] O. Nierstrasz, M. Denker, T. Gîrba, and A. Lienhard. Analyzing, capturing and taming software change. In *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.

[28] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, Oct. 1992.

[29] F. Pluquet and R. Wuyts. Evolution persistence for objects. In *Proceedings of EVOL 2006 (1st International ERCIM Workshop on Challenges in Software Evolution)*, pages 155–158, 2006.

[30] 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 Computer Society, 2005.

[31] R. Robbes and M. Lanza. An approach to software evolution based on semantic change. In *Proceedings of FASE 2007*, pages 27–41, 2007.

[32] R. Robbes and M. Lanza. A change-based approach to software evolution. In *ENTCS volume 166, issue 1*, pages 93–109, 2007.

[33] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *Proceedings of ICPC 2007*, page to appear, 2007.

[34] D. Roberts and J. Brant. Tools for making impossible changes - experiences with a tool for transforming large smalltalk programs. *IEE Proceedings - Software*, 152(2):49–56, Apr. 2004.

[35] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[36] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: Raising awareness among configuration management workspaces . In *ICSE*, pages 444–454. IEEE Computer Society, 2003.

[37] T. Schümmer and J. M. Haake. Supporting distributed software development by modes of collaboration. In W. Prinz, M. Jarke, Y. Rogers, K. Schmidt, and V. Wulf, editors, *ECSCW*, pages 79–98. Kluwer, 2001.

[38] J. Sutherland. Business objects in corporate information systems. *ACM Computing Surveys*, 27(2):274–276, 1995.

[39] G. M. Weinberg. *The Psychology of Computer Programming*. Dorset House, silver anniversary edition edition, 1998.

[40] R. Wettel and R. Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Proceedings of SYNASC 2005*, pages 63–70, 2005.

[41] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

[42] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM Press.

[43] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, oct 2005.

[44] P. Zumkehr. Changeboxes — modeling change as a first-class entity. Master's thesis, University of Bern, Feb. 2007.