An Approach to Software Evolution Based on Semantic Change

Romain Robbes *and* Michele Lanza *and* Mircea Lungu

Faculty of Informatics
University of Lugano, Switzerland

Abstract. The analysis of the evolution of software systems is a useful source of information for a variety of activities, such as reverse engineering, maintenance, and predicting the future evolution of these systems.
Current software evolution research is mainly based on the information contained in versioning systems such as CVS and SubVersion. But the evolutionary information contained therein is incomplete and of low quality, hence limiting the scope of evolution research. It is incomplete because the historical information is only recorded at the explicit request of the developers (a *commit* in the classical checkin/checkout model). It is of low quality because the file-based nature of versioning systems leads to a view of software as being a set of files.
In this paper we present a novel approach to software evolution analysis which is based on the recording of *all* semantic changes performed on a system, such as refactorings. We describe our approach in detail, and demonstrate how it can be used to perform fine-grained software evolution analysis.

1 Introduction

The goal of software evolution research is to use the history of a software system to analyse its present state and to predict its future development [1] [2]. It can also be used to complement existing reverse engineering approaches to understand the current state of a system [3] [4] [5] [6]. The key to perform software evolution research is the quality and quantity of available historical information. Traditionally researchers extract historical data from versioning systems (such as CVS and SubVersion), which at explicit requests by the developers record a snapshot of the files that have changed (this is widely known as the checkin/checkout model).

We argue that the information stored in current versioning systems is not accurate enough to perform higher quality evolution research, because they are not explicitely designed for this task: Most versioning systems have been developed in the context of software configuration management (SCM), whose goal is to manage the evolution of large and complex software systems[7]. But SCM serves different needs than software evolution, it acts as a *management support discipline* concerned with controlling changes to software products and as a *development support discipline* assisting developers in performing changes to software products[8] [9]. Software evolution on the other hand is concerned with the phenomenon of the evolution of software itself. The dichotomy between SCM and software evolution has led SCM researchers to consider software evolution research as a mere “side effect” of their discipline [10].

Because most versioning systems originated from SCM research, the focus has never been on the quantity and quality of the recorded evolutionary information, which we consider as being (1) insufficient and (2) of low quality. It is insufficient because information only gets recorded when developers commit their changes. In previous work [11] we have analyzed how often developers of large open-source projects commit their changes and found that the number of commits per day barely surpasses 1 (one commit on average every 8 "working day" hours). The information is of low quality because there is a loss of semantic information about the changes: only textual changes get recorded. For example, to detect structural changes such as refactorings one is forced to tediously reconstruct them from incomplete information with only moderate success[12] [13]. Overall this has a negative impact on software evolution research whose limits are set by the quality and quantity of the available information.

This paper presents our approach to facilitate software evolution research by the accurate recording of *all semantic changes* that are being performed on a software system. To gather this change information, we use the most reliable source available, namely the Integrated Development Environment (IDE, such as Eclipse [1] or Squeak [2]) used to develop object-oriented software systems.

Modern development environments allow programmers to perform semantic actions on the source code with ease, thanks to semi-automatic refactoring [14] support. They also have an open architecture that tools can take advantage of: The event notification system the IDE uses can be monitored to keep track of how the developers modify the source code. From this information, we build a model of the evolution of a system in which the notion of *change* takes on a primary role, since people develop a software system by incrementally changing it [15]. The notion of incremental change is further supported by IDEs featuring incremental compilation where only the newly modified parts get compiled, *i.e.,* an explicit system building phase where the whole system is being built from scratch is losing importance.

In our model, the evolution of a system is the sequence of changes which were applied to develop it. These changes are operations on the program's abstract syntax tree at the simplest level. Through a composition mechanism, changes are grouped to represent larger changes associated with a semantic meaning, such as method additions, refactorings, feature additions or bug fixes. Thus we can reason about a system's evolution on several levels, from a high-level view suitable to a manager down to a concrete view suitable to a developer wishing to perform a specific task.

We store the change information in a repository, to be exploited by tools integrated in the IDE the programmer is using. After presenting our approach, we show preliminary results, based on the *change matrix*, an interactive visualization of the changes applied to the system under study.

Structure of the paper. Section 2 presents the principles and a detailed overview of our approach. Section 3 presents a case study we performed to validate our approach, in which we used the change matrix visualization to assess the evolution of projects done by students. Section 5 and 4 compare our approach to more traditional approaches to

[1] http://www.eclipse.org
[2] http://www.squeak.org

software evolution analysis. Section 6 briefly covers the implementation. In Section 7 we conclude and outline future work.

2 Change-based Object-Oriented Software Evolution

Our approach to software evolution analysis is based on the following principles:

- *Programming is more than just text editing*, it is an incremental activity with semantics. If cutting out a piece of a method body and wrapping it into its own method body can be seen as cut&paste, it is in fact an *extract method* refactoring. Hence, instead of representing a system's evolution as a sequence of versions of text files, we want to represent it as a sequence of explicit changes with object-oriented semantics.
- *Software is in permanent evolution*. Modern Integrated Development Environments (IDE), such as Eclipse, are a very rich and accurate source of information about a system's life-cycle. IDEs thus can be used to build a change-based model of evolving object-oriented software and to gather the change data, which we afterwards process and analyze. Based on the analyzed data, we can also create tools which feed back the analyzed data into the IDE to support the development process.

Taming Change. Traditional approaches to evolution analysis consider the history of a system as being a sequence of program versions, and compute metrics or visualize these versions to exploit the data contained in them[16][17]. Representing evolution as a sequence of version fits the format of the data obtained from a source code repository. There is a legitimate doubt that the nature of existing evolution approaches is a direct consequence of the representation adopted by versioning systems, and is therefore limited by this.

The phenomenom of software evolution is one of *continuous change*. It is not a succession of program versions. Our approach fits this view because it models the evolution of a software system as a sequence of changes which have inherent object-oriented semantics, focusing on the phenomenon of change itself, rather than focusing on the way to store the information. We define semantic changes as changes at the design level, not at the behavioral level as in [18].

Modeling software evolution as meaningful change operations fits the inherently incremental nature of software development, because this is the very way with which developers are building systems. Programmers modify software by adding tiny bits of functionality at a time, and by testing often to get feedback. At a higher level features and bug fixes are added incrementally to the code base and at an even higher level the program incrementally evolves from one milestone version to another.

A consequence of this approach is that by recording only the changes we do not explicitly store versions, but we can reconstruct any version by applying the changes. In SCM this concept is called change-based versioning [9], however the fundamental difference between our approach and the existing ones is that the changes in our case feature fine-grained object-oriented semantics and are also first-level executable entities.

Our goal is to build a model of evolution based on a scalable representation of change. First we discuss how we represent programs, then we examine how we model changes and how we extract them from IDEs.

Representing Programs. Our model defines the history of a program as the sequence of changes the program went through. From these changes we can reconstruct each successive state of the program source code. We represent one state of the entire program as one abstract syntax tree (AST). Below the root are the packages or modules of the program. Each package in turn has children which are the package's classes. Class nodes have children for their attributes and their methods. Methods also have children. The children of a method form a subtree which is obtained by parsing the source code contained in the method. Thus each entity of a program, from the package level down to the program statement level, is represented as a node in the program's AST, as show in Figure 1.

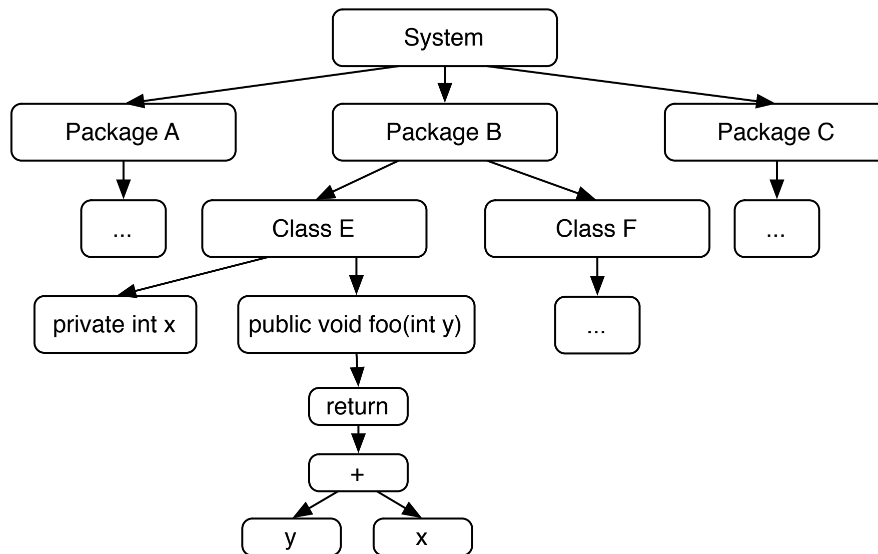

Fig. 1. We represent a state of a program as an abstract syntax tree (AST).

Each node contains additional information that is stored in properties associated with the node. The set of properties (and their values) defined on a node can be seen as its label or meta-information. Properties depend on the type of nodes they are associated to.

For instance, class nodes have properties like *name*, *superclass*, and *comment*. An instance variable has a *name* property. In a statically typed programming language it

would also have a *type* property, as well as a *visibility modifier* in the case of Java. Methods have a *name* and could have properties encoding its type signature in a typed language. The property system is open so that other properties can be added at will.

Extracting the Changes. The type of semantic change information that we model is not retrievable from existing versioning systems[19]. Such detailed information about a software system can be retrieved from the IDEs developers are using to build software systems. IDEs are a good source of information because:

- They feature a complete model of a program to provide advanced functionalities such as code completion, code highlighting, navigation facilities and refactoring tools. Such a model goes beyond the representation at the file level to reference program-level entities such as methods and classes in an object-oriented system.
- They feature an event notification system allowing third-party tools to be notified when the user issues a change to the program. Mechanisms such as incremental compilation and smart completion of entity names take advantage of this.
- IDEs allow a user to automatically perform high-level transformations of programs associated with a semantic meaning, namely refactorings. These operations are easy to monitor in an IDE, but much harder to detect outside of it, since they are lost when the changed files are committed to a software repository [12] [13].

Since some IDEs are extensible by third parties with plugin mechanisms, our tools can use the full program model offered by the IDEs to locate and reason about every statement in the program, and can be notified of changes *without relying on explicit action by the developers*. This mechanisms alleviate the problems exhibited by the use of versioning systems: It is easier to track changes applied to an entity in isolation rather than attempting to follow it through several versions of the code base, each comprising a myriad of changes. Furthermore, after each notification, the IDE can also be queried for time and author information.

Representing the Changes. We model changes as first-class executable entities. It is possible to take a sequence of changes and execute it to build the version of the program represented by them. Changes can also be reversed (or undone) to achieve the effect of going back in time. Changes feature precise time and authorship information, allowing the order of the changes to be maintained. In contrast, most other approaches reduce the time information to the time where the change was checked in, following the checkin/checkout model supported by the versioning systems, such as CVS and SubVersion.

There are two distinct kinds of changes, (1) low-level changes operating at the syntactic level, and (2) higher-level changes with a semantic meaning, which are composed of lower-level changes.

1. **Syntactic Changes.** They are simple operations on the program AST, defined as follows:
   - A *creation* creates a node n of type t, without inserting it into the AST.

   - An *addition* adds a node n to the tree, as a child of another specified node m. If order is important, an index can be provided to insert the node n in a particular position in the children of m. Otherwise, n is appended as the last child of m.
   - A *deletion* removes the specified node n from its parent m.
   - A *property change* sets the property p of node n to a specific value v.

   Using these low-level changes, we view a program as an evolving abstract syntax tree. A program starts as an empty tree and an empty change history. As time elapses, the program is built and the AST is populated. At the same time, all the change operations which were performed to build the program up to this point are stored in the change history.
2. **Semantic Changes.** To reason about a system, we need to raise the level of abstraction beyond mere syntactic changes. This is achieved by the composition mechanism. A sequence of lower-level changes can be composed to form a single, higher level change encapsulating a semantic meaning. Here are a few examples:
   - A sequence of consecutive changes involving a single method m can be interpreted as a single method *implementation*, or *modification* if m already existed.
   - Changes to the structure of a class c (attributes, superclass, name) are either a class *definition* or a class *redefinition*, if c existed before the changes. These kinds of changes form the *intermediate* changes.
   - At a higher level, some sequences of intermediate changes are *refactorings*[14]. They can be composed further to represent these higher-level changes to the program. For example, the "extract method" refactoring involves the *modification* of a first method $m1$ (a sequence of statements in $m1$ is replaced by a single call to method $m2$), and the *implementation* of $m2$ (its body comprises the statements that were removed from $m1$). In the same way, a "rename class" refactoring comprises the *redefinition* of the class (with a name change), and the *modification* of all methods because of the changed referenced class name.
   - We define a *bug fix* as the sequence of intermediate changes which were involved in the correction of the faulty behavior.
   - In the same way, a *feature* implementation is comprised of all the changes that programmers performed to develop the feature. These changes can be intermediate changes as well as any refactorings and bug fixes which were necessary to achieve the goal.
   - At an even higher level, we can picture *main program features* as being an aggregation of smaller features, and *program milestones* (major versions) as a set of high-level features and important bug fixes.

The composition of changes works at all levels, to allow changes to represent higher-level concepts. This property is a key point to the scalability of our approach. Without it, we would have to consider only low-level, syntactic changes, and hence be limited to trivial programs, because of the sheer quantity of changes to consider. In addition to composition, it is also possible to analyse the evolution of a system by considering subsets of changes. Thus a high-level analysis of a system would only take into account the changes applied to classes and packages, in order to have a bird's eye view of the system's evolution. The lower-level changes are still useful to analyse the evolution: Once an anomaly has been identified in a high-level strata of the system, lower-level

changes can be looked at to infer the particular causes of a problem. For example, if a package or a module of the system needs reengineering, then its history in terms of classes and methods can be summoned. Once the main culprits of the problem have been identified, these few classes can be viewed in even more detail by looking at the changes in the implementation of their methods.

To sum up, we consider the program under analysis as an evolving abstract syntax tree. We store in our model all the change operations necessary to recreate the program at any point in time. At the lowest level, these operations consist of creation, addition and removal of nodes in the tree, and of modifications of node properties. These changes can be composed to represent higher-level changes corresponding to actions at the semantic level, such as refactorings, bug fixes *etc.*

3 Case Studies

Since our approach relies on information which was previously discarded, we can not use existing systems as case studies. We monitored new projects to collect all the information. Our case studies are projects done by students over the course of a week. These projects are small (15 to 40 classes), but are interesting case studies since the code base is foreign to us. There were 3 possible subjects to choose from: A virtual store in the vein of Amazon (Store), a simple geometry program (Geom), and a text-based role-playing game (RPG). Table 1 shows a numerical overview of the projects we have tracked (each project is named with a letter, from A to I). The frequency of the recorded changes was very high compared to a that of a classical versioning system: While the projects lasted one week, their actual coding time was in the range of hours. Considering this fact and that the students were novice programmers, our approach allows for an unprecedented precision with respect to the recording of the evolution.

Project	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
Type	**Geom**	**Store**	**Store**	**Store**	**Store**	**RPG**	**Geom**	**Store**	**RPG**
Class Added	22	14	14	9	12	15	21	12	41
Class Modified	65	17	34	13	6	24	57	15	27
Class Commented	0	12	0	0	1	0	0	0	0
Class Recategorized	0	0	5	0	0	0	0	0	11
Class Renamed	0	0	0	0	1	1	0	0	1
Class Removed	10	1	5	5	0	3	6	2	18
Attributes Added	82	19	29	19	20	61	30	29	137
Attributes Removed	50	7	13	5	2	19	15	5	54
Method Added	366	119	182	164	117	237	219	135	415
Method Modified	234	69	117	140	81	154	143	118	185
Method Removed	190	20	81	32	13	38	117	21	106

Table 1. A numerical overview of the semantic changes we recovered from the projects.

The changes considered here are *intermediate-level* changes, one per semantic action the user did (in that case, mainly class and method modifications: the students were familiar with refactoring). The table classifies the changes applied to each project. We can already see some interesting trends: Some projects have a lot more “backtracking”

(removals of entities) than others; usage of actions related to refactoring (commenting, renaming, repackaging entities) varies widely between projects.

In the remainder of the section, we concentrate on the analysis of one of the projects, namely the role-playing game project I (the last column of the table). More details on the other projects are available in the extended version of [11].

3.1 Detailing the Evolution of a Student Project

We chose project I for a detailed study, because it had the most classes in it, and was the second largest in statements. Project I is a role-playing game in which a player has to choose a character, explore a dungeon and fight the creatures he finds in it. In this process, he can find items to improve his capabilities, as well as gaining experience.

We base our analysis on the change matrix Figure 2 inspired by [17]. It is a timeline view of the changes applied to the entire system, described in terms of classes and methods (a coarser-grained version, displaying packages and classes is also available, but not shown in this paper).

The goal of the change evolution matrix is to provide the user with an overview of the activity in the project at the method level granularity over time. Time is mapped on the x-axis. Every method is allocated a horizontal band which is gray for the time period in which the method existed and white otherwise. The method bands are grouped by classes, and ordered by their creation time. Classes are delimited by black lines and are also ordered by their creation time, with the oldest classes at the top of the figure.

Changes are designed by colors: green for the creation of a method, blue for its removal and orange for a modification. Selecting a change shows the method's source code after the change is applied to the system. A restriction of the figure at this time of writing is that it does not show when a class is deleted.

Figure 2 is rotated for increased readability. Events are mapped on intervals lasting 35 minutes. Note that to ease comprehension the system size is reported on the left of the page, and sessions are delimited by rectangles with rounded corners in both the matrix and the graph size view. Also, the class names are indicated below the figure. Figure 3 represent the same matrix, but focused on the class Combat. Since its lifespan is shorter, we can increase the resolution to five minutes per interval.

Considering the classes and their order of creation (Figure 2), we can see that the first parcels of functionality were, in order: The characters; the weapons; the enemies; the combat algorithm; the healing items and finally the dungeon itself, defined in terms of rooms. We can qualify this as a bottom-up development methodology.

After seeing these high-level facts about the quality-wise and methodology-wise evolution of the system, we can examine it session by session. Each session has been identified visually and numbered. Refer to Figure 2 to see the sessions.

Session 1, March 27, afternoon: The project starts by laying out the foundations of the main class of the game, Hero. As we see on the change matrix, it evolves continually throughout the life of the project, reflecting its central role. At the same time, a very simple test is created, and the class Spell is defined.

Session 2, March 28, evening: This session sees the definition of the core of the character functionality: Classes Hero and Spell are changed, and classes Items,

Fig. 2. Change matrix of project I.

Mage, Race and Warrior are introduced, in this order. Since Spells are defined, the students define the Mage class, and after that the Warrior class as another subclass of Hero. This gives the player a choice of profession. The definitions are still very shallow at this stage, and the design is unstable: Items and Race will never be changed again after this session.

Session 3, March 28, night: This session supports the idea that the design is unstable, as it can be resumed as a failed experiment: A hierarchy of races has been introduced, and several classes have been cloned and modified (Mage2, Hero3 *etc.*). Most of these classes were quickly removed.

Session 4, March 29, afternoon: This session is also experimental in nature. Several classes are modified or introduced, but were never touched again: Hero3, CEC, RPGCharacter (except two modifications later on, outside real coding sessions). Mage and Warrior are changed too, indicating that some of the knowledge gained in that experiment starts to go back to the main branch.

Session 5, March 29, evening and night: This session achieves the knowledge transfer started in session 4. Hero is heavily modified in a short period of time, while Mage and Warrior are consolidated.

Session 6, March 30, late afternoon: This session sees a resurgence of interest for the offensive capabilities of the characters. A real Spell hierarchy is defined (Lightning, Fire, Ice), while the Weapons class is slightly modified as well.

Session 7, March 31, noon: The first full prototype of the game. The main class, RPG (standing for Role Playing Game) is defined, as well as an utility class called Menu. Mage, Warrior and their superclass Hero are modified.

Session 8, March 31, evening: This session consolidates the previous one, by adding some tests and reworking the classes changed in session 7.

Session 9, March 31, night: This session focuses on weapon diversification with classes Melee and Ranged; these classes have a very close evolution for the rest of their life, suggesting some data classes. At the same time, a real hierarchy of hostile creatures appears: Enemies, Lacche, and Soldier. The system is a bit unstable at that time, since Enemies experiences a lot of method which were added then removed immediately, suggesting renames.

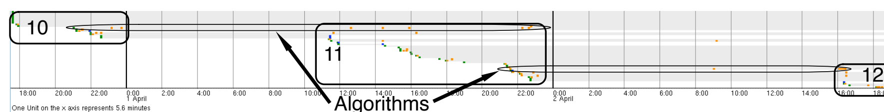

Fig. 3. Change matrix zoomed on the class Combat

Session 10, April 1st, noon to night: This intensive session sees the first iteration of the combat engine. The weapons, spells and characters are first refined. Then a new enemy, Master, is defined. The implementation of the Combat class shows a lot of modifications of the Weapon and Hero classes. An Attack class soon appears. Judg-

ing from its (non-)evolution, it seems to be a data class with no logic. After theses definitions, the implementation of the real algorithm begins. We see on Figure 3 –the detailed view of combat– that one method is heavily modified continuing in the next session.

Session 11, April 2, noon to night: Development is still heavily focused on the Combat algorithm. Classes of Potion and Healing are also defined, allowing the heroes to play the game for a longer time. This session also modifies the main combat algorithm, and at the same time, two methods in the Hero class, showing a slight degree of coupling. A second method featuring a lot of logic is implemented, as shown in Figure 3: several methods are often modified.

Session 12: April 3, afternoon to night: This last session finishes the implementation of Combat –changing the enemy hierarchy in the process–, and resumes the work on the entry point of the game, the RPG class. Only now is a Room class introduced, providing locality. These classes are tied to Combat to conclude the main game logic. To finish, several types of potions –simple data classes– are defined, and a final monster, a Dragon, is added at the very last minute.

4 Discussion

Compared to traditional approaches, extracting information from source control repositories, our change-based approach has a number of advantages (accurate information, scalable representation, and version generation), but also some limitations (portability, availability of case studies, and performance).

- **Accurate information.** The information we gather is more accurate in several ways. It consists of program-level entities, not mere text files which incurs extra treatment to raise the level of abstraction. Since we are notified of changes in an automatic, rather than explicit way, we can extract finer change information: Each change can be processed *in context*. The time information we gather is accurate up to the second, whereas a versioning system reduces it to the checkin time. Processing changes in context and in a timely manner allows us to track entities through their life time while being less affected by system-wide changes such as refactorings.
- **Scalable representation.** We represent every statement of a system as separate entities, and every operation on those statements as a first-class change operation. Such a precise representation enables us to reflect on very focused changes, during defined time period and on a distinct set of low-level entities. At the other end of the spectrum, changes can be composed into semantic level changes such as method modifications, class additions, or even entire sessions, while the entities we reflect on can be no longer statements, but methods, classes or packages. Thus our approach can both give a “big picture” view to a manager, as well as a detailled summary of the changes submitted by a developer during his or her last coding session.
- **Version generation.** Since changes are executable, we can also reproduce versions of the program. We can thus revert to version analysis and more traditional approaches when we need to.

– **Portability.** Our approach is currently both language-specific and environment-specific. This allows us to leverage to the maximum the properties of the target language and the possibilities offered by the IDE (in our case, Smalltalk and Squeak). However, it implies a substantial porting effort to use our approach in another context. Consequently, one of our goal is to extract the language and environment-independent concepts to ease this effort. Thus we will port our prototype to the Java/Eclipse platform. The differences in behavior between the two versions will help us isolate the common concepts.
– **Availability of case studies.** As mentioned above, we can not use pre-existing projects as case studies since we require information which was discarded previously. Solving this problem is one of our priorities. Beyond using student projects as case studies, we are monitoring our prototype itself for later study. This would be a medium-sized case study: At the time of writing, it comprised 203 classes and 2249 methods over 11681 intermediate changes. We also plan to release and promote our tools to the Smalltalk community (the language our tools are implemented in) soon. In the longer term, porting our tools to the Eclipse platform will enable us to reach a much wider audience of developers.
– **Performance.** Our approach stores operations rather than states of programs. The large number of changes and entities could raise performance concerns. It takes around one minute to generate all the possible versions of our prototype itself from the stored changes. The machine used was a 1.5 GHz portable computer, our prototype having around 11'000 intermediate changes.

5 Related Work

Several researchers have analysed the evolution of large software systems, basing their work on system versions typically extracted from software repositories such as CVS and SubVersion [20] [21] [22] [23] [24]. In most cases these approaches have specific analysis goals, such as detecting logical couplings [25] or extracting evolutionary patterns [4].

Several researchers raised the abstraction level beyond files to consider design evolution. In [22], Xing and Stroulia focus on detecting evolutionary phases of classes, such as rapidly developing, intense evolution, slowly developing, steady-state, and restructuring. They had to sample their data for their case study and used only the 31 minor versions of the project. Parsing and analysing the 31 versions took around 370 minutes on a standard computer, which rules out an immediate use by a developer. [26] presents a methodology to connect high-level models to source code, but has only been applied to a single version of a system so far. [23] describes how hierarchies of classes evolve, but still depends on sampling and the checkin/checkout model. [20] applies origin analysis to determine if files moved between versions. In [18], Jackson and Ladd present an approach to differencing C programs at the semantic level. They define semantic changes as dependency changes between inputs and outputs, while we are primarily interested in design-level changes.

All these and other known approaches cannot perform a fine-grained analysis because the underlying data is restricted by the data that can be extracted from versioning

system, tying them to the *checkin/checkout* model. In [11], we outlined the limitations of this model to retrieve accurate evolutionary information. Versioning systems restrain their interactions with developers to explicit retrieval of the source (check out), and submission of the modified sources once the developer finishes his task (check in or commit). All the changes to the code base are merged together at commit time, and become hard to distinguish from each other. The time stamp of each modification is lost, and changes such as refactorings become very hard, if not impossible to detect. Even keeping track of the identity of a program element can be troublesome if it has been renamed.

Moreover, most versioning systems version text files. This guarantees language-independence but limits the quality of the information stored to the lowest common denominator: An analysis of the system's evolution going deeper than the file level requires the parsing of (selected) versions of the system and the linking of the successive versions. Such a procedure is costly [19]. Thus it is a common practice to first *sample* the data, by only retaining a fraction of the available versions. The differences between two versions retained for analysis becomes even larger, so the quality of the data degrades further.

Mens [27] presents a thorough survey of merging algorithms in versioning systems, of which [28] is the closest to our approach: operations performed on the data are used as the basis of the merging algorithm, not the data itself. However, the operations are not precised in the paper and are used only in the merge process. The change mechanism used by Smalltalk systems uses the same idea, but the changes are not abstracted. Smalltalkers usually don't rely on them and use more classic, state-based versioning systems. In addition, most of the versioning systems covered by Mens are not used widely in practice: most evolution analysis tools are based on the two most used versioning systems, CVS and SubVersion.

6 Tool Implementation

Our ideas are implemented for the Smalltalk language and the Squeak IDE in SpyWare, shown in Figure 4. From top to bottom, we see: the main window; a code browser on a version of project I; the change matrix of project I; and a graph showing the growth rate of the system.

7 Conclusion and Future Work

We presented a fine grained, change-based approach to software evolution analysis and applied it to nine student projects, one of which was analyzed in detail. Our approach considers a system to be the sequence of changes that built it, and extract this information from the IDE used during development. We implemented this scheme and performed an evolution analysis case study based on a software visualization tool –the change matrix– we built on top of this platform.

Although our results are still in their infancy, they are encouraging as they allow us to focus on particular entities in a precise period of time once a general knowledge of the system has been gained. In our larger vision, we want a more thorough interaction of

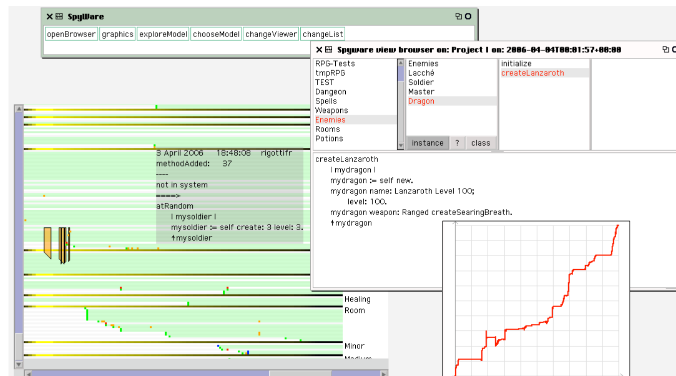

Fig. 4. Screen capture of SpyWare, our prototype

forward and reverse engineering to support rapidly changing systems. In this scenario, developers need this detailed analysis of part of the system as much as they need a global view of the systems' evolution.

We have only scratched the surface of the information available in these systems. We plan to use more advanced tools, visualizations, and methods (such as complexity metrics) to meaningfully display and interact with this new type of information, and envision other uses beyond evolution analysis.

References

1. Lehman, M., Belady, L.: Program Evolution: Processes of Software Change. London Academic Press, London (1985)
2. Gall, H., Jazayeri, M., Klösch, R., Trausmuth, G.: Software evolution observations based on product release history. In: Proceedings International Conference on Software Maintenance (ICSM'97), Los Alamitos CA, IEEE Computer Society Press (1997) 160–166
3. Mens, T., Demeyer, S.: Future trends in software evolution metrics. In: Proceedings IWPSE2001 (4th International Workshop on Principles of Software Evolution). (2001) 83–86
4. Van Rysselberghe, F., Demeyer, S.: Studying software evolution information by visualizing the change history. In: Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04), Los Alamitos CA, IEEE Computer Society Press (2004) 328–337
5. Gîrba, T., Ducasse, S., Lanza, M.: Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In: Proceedings 20th IEEE International Conference on Software Maintenance (ICSM 2004), Los Alamitos CA, IEEE Computer Society Press (2004) 40–49
6. D'Ambros, M., Lanza, M.: Software bugs and evolution: A visual approach to uncover their relationship. In: Proceedings of CSMR 2006 (10th IEEE European Conference on Software Maintenance and Reengineering), IEEE Computer Society Press (2006) 227 – 236
7. Tichy, W.: Tools for software configuration management. In: Proceedings of the International Workshop on Software Version and Configuration Control. (1988) 1–20
8. Feiler, P.H.: Configuration management models in commercial environments. Technical report cmu/sei-91-tr-7, Carnegie-Mellon University (1991)

9. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Computing Surveys **30**(2) (1998) 232–282
10. Estublier, J., Leblang, D., van der Hoek, A., Conradi, R., Clemm, G., Tichy, W., Wiborg-Weber, D.: Impact of software engineering research on the practice of software configuration management. ACM Transactions on Software Engineering and Methodology **14**(4) (2005) 383–430
11. Robbes, R., Lanza, M.: A change-based approach to software evolution. In: ENTCS volume 166. (2007) to appear
12. Görg, C., Weissgerber, P.: Detecting and visualizing refactorings from software archives. In: Proceedings of IWPC (13th International Workshop on Program Comprehension, IEEE CS Press (2005) 205–214
13. Filip Van Rysselberghe, M.R., Demeyer, S.: Detecting move operations in versioning information. In: Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR'06), IEEE Computer Society (2006) 271–278
14. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
15. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley (2000)
16. Gîrba, T., Lanza, M., Ducasse, S.: Characterizing the evolution of class hierarchies. In: Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005), Los Alamitos CA, IEEE Computer Society (2005) 2–11
17. Lanza, M.: The evolution matrix: Recovering software evolution using software visualization techniques. In: Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution). (2001) 37–42
18. Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In Müller, H.A., Georges, M., eds.: ICSM, IEEE Computer Society (1994) 243–252
19. Robbes, R., Lanza, M.: Versioning systems for evolution research. In: Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution), IEEE Computer Society (2005) 155–164
20. Tu, Q., Godfrey, M.W.: An integrated approach for studying architectural evolution. In: 10th International Workshop on Program Comprehension (IWPC'02), IEEE Computer Society Press (2002) 127–136
21. Jazayeri, M., Gall, H., Riva, C.: Visualizing Software Release Histories: The Use of Color and Third Dimension. In: Proceedings of ICSM '99 (International Conference on Software Maintenance), IEEE Computer Society Press (1999) 99–108
22. Xing, Z., Stroulia, E.: Analyzing the evolutionary history of the logical design of object-oriented software. IEEE Trans. Software Eng. **31**(10) (2005) 850–868
23. Gîrba, T., Lanza, M.: Visualizing and characterizing the evolution of class hierarchies (2004)
24. Eick, S., Graves, T., Karr, A., Marron, J., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE Transactions on Software Engineering **27**(1) (2001) 1–12
25. Gall, H., Hajek, K., Jazayeri, M.: Detection of logical coupling based on product release history. In: Proceedings International Conference on Software Maintenance (ICSM '98), Los Alamitos CA, IEEE Computer Society Press (1998) 190–198
26. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software reflexion models: Bridging the gap between design and implementation. IEEE Trans. Software Eng. **27**(4) (2001) 364–380
27. Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering **28**(5) (2002) 449–462
28. Lippe, E., van Oosterom, N.: Operation-based merging. In: SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments, New York, NY, USA, ACM Press (1992) 78–87