Characterizing and Understanding Development Sessions

Romain Robbes and Michele Lanza
Faculty of Informatics, University of Lugano - Switzerland

Abstract

The understanding of development sessions, the phases during which a developer actively modifies a software system, is a valuable asset for program comprehension, since the sessions directly impact the current state and future evolution of a software system. Such information is usually lost by state-of-the-art versioning systems, because of the checkin/checkout model they rely on: a developer must explicitly commit his changes to the repository. Since this happens in arbitrary and sometimes long intervals, recovering the changes between two commits is difficult and inaccurate, and recovering the order of the changes is impossible.

We have implemented an evolution monitoring prototype which records every *semantic change performed on a system, and is able to completely reconstruct development sessions. In this paper we use this fine-grained information to understand and characterize the development sessions as they were carried out on two object-oriented systems.*

1 Introduction

The field of software evolution [8] uses the history of software as an asset to understand its actual state [3, 5] and to predict its future development [9, 18]. Traditionally, the history is extracted from the repositories of source code control systems, such as RCS, CVS and SubVersion, which contain valuable information about a system's evolution. Yet they do not contain all the information about an evolving software system. In particular, due to the checkin-checkout model such systems rely on, they only store data when a developer explicitly commits a set of changes to them, *e.g.,* when (s)he finishes a task or when the working day is over. Thus the versioning systems store only arbitrarily spaced snapshots of the system's state, and not all the intermediate steps. We already outlined the limitations of this approach for software evolution analysis in previous work [12], [13]: Not recording the intermediate modifications to the code between two commits leads to decreased information quality. Changes become hard to tell apart from each other if they happen at the same location and/or at the same time. In addition, the versioning of mere files implies a computationally expensive pre-processing to reconstruct the changes (such as refactorings [6]) performed on the source code.

In our previous work [14] we presented an approach to record *all* the semantic changes that are being performed on a system, by monitoring the integrated development environment (IDE). Modern IDEs such as Eclipse, supporting incremental compilation and refactoring engines, have led to the perception that instead of being written, software is being *constructed* by applying sequentially local changes to a part of the system, which is then incrementally compiled.

In this paper we focus on the information that our approach records, but which cannot be recovered *at all* from a state-of-the-art versioning system repository: The exact sequence of events forming a development session. Versioning systems store only the final result committed by the developer, and have no way to tell the order in which changes happened during a session. We argue that the sequence of events happening during a development session can lead to valuable insights on the way the system is being constructed, which in turn are useful for program comprehension (who did what and when?) and reverse engineering tasks (what led a certain part of the system to be like this? What were the decisions and the steps taken?). This information is also useful to understand the phenomenon of evolution itself (how do programmers write code? how long are the sessions? are there different types of sessions? are there development patterns that a developer follows?). To demonstrate our approach, we analyzed hundreds of development sessions carried out on two object-oriented systems written in Smalltalk.

Structure of the paper. Section 2 presents change-based evolution in a nutshell. Section 3 shows with a motivating example why understanding development sessions is useful. Section 4 enumerates the characteristics we defined and looked for in our case studies. Section 5 presents our two case studies and shows through simple visualizations selected development sessions described in detail. Section 7 presents our tool set, while Section 6 discusses our approach and related work. Finally, Section 8 concludes the paper.

2 Change-based Evolution In a Nutshell

We model software evolution as a sequence of changes that take a system from one state to the next by means of semantic transformations [14, 13]. By semantic transformation we mean an incremental compilation step offered by IDEs such as Eclipse, where the developer is incrementally applying modifications on specific parts of the system. 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 rather as the sum of *change operations* which brought the system to its actual state. With historical information such as the one offered by CVS or SubVersion, these operations can not be deduced with enough precision by differencing two arbitrary program versions [7] once the evolution has happened. Instead we recover the semantic changes by monitoring the IDE while programmers are building the software.

2.1 Program Representation

Our approach represents programs as domain-specific entities rather than text files. Since we focus on object-oriented programs, we store and analyse constructs such as classes and methods, not files and lines. We represent a software system as an evolving abstract syntax tree (AST) containing nodes which represent packages, classes, methods, variables and statements. A node *a* is a child of a node *b* if *a* contains *b* (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.*

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.

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 transition from one state of the evolving system to the next. Some examples of change operations are: adding/removing class/methods to/from the system, changing the implementation of a method, or refactorings. We support *atomic* and *composite* change operations.

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 childs 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, adding or changing 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 [2]. 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: A 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 Motivating Example

Alice is working for a parcel delivery company, where she is building a parcel tracking application. There are several kinds of parcels depending on various shipping requirements. Each parcel can contain several items; parcels are shipped in trucks, and/or planes. We are confronted with a tree structure of elements of various types, all contained in the Vehicle and Parcel class hierarchies. A report printing feature has been implemented. Alice needs to implement rules to check if parcels have a valid address and are dispatched appropriately to the delivery trucks.

Session A: A failed experiment. Alice starts two distinct implementations with numerous methods scattered around two class hierarchies. Alice quickly gets lost, and because of the difficulties, she deletes the first implementation and keeps the second one, which is however incomplete and buggy. Alice commits to the project's CVS repository.

Session B: Towards a solution. During this second session, Alices sees the opportunity to introduce a visitor pattern [4] to both define the address rules and to refactor the printing functionality. The visitor allows through double-dispatch to move functionality scattered in several classes into a single class. Alice proceeds as follows:

B1: She implements the basic visitor with a general AbstractVisitor class and a double-dispatch mechanism. She then defines the dispatch methods for each class (Truck, Plane, Parcel and its subclasses, Item, *etc.*).

B2: She implements a basic visitor to test the visitor implementation (it prints the name of the classes it traverses to the standard output).

B3: Alice then moves the report printing feature to the new implementation. She has to slightly change its interface and the tests associated to it.

B4: Finally, she re-implements the address checking feature in another visitor subclass and this time succeeds in implementing it. She reuses part of the implementation from session A, and deletes the rest.

Alice again commits again to the project CVS. Bob, her office mate, wants to review what Alice has done. He checks out the current state of the project. All that Bob can gather is that the code base has been changed twice, but all the decisions, the design steps, *etc.* are lost.

Let us highlight a few scenarios in which a precise characterization of the development sessions is useful:

Focusing the Reviewing and Testing Effort. Code reviewing is an established practice to detect defects. Characterizing development sessions could highlight sessions with non-standard features and hence guide the review process to allocate more resources to code originating from sessions with a higher risk of containing bugs. In our example, session A should be checked carefully. Code modified during session B but introduced during session A should be checked more than the code pertaining to the refactorings.

Fine-grained Comprehension of Sessions. Once a coding session has been selected either for reviewing or reverse engineering, it needs to be understood. Having a complete timeline of its events helps in understanding the outcome of a development session for several reasons:

- The characterization information gives a general context to the session: A refactoring-dominant session is different from a bug-fixing, feature addition or feature enhancement session.
- Reviewing events in time eases their understanding. Events usually follow the logical steps a developer takes: Methods depending on each other are implemented and modified at the same time. On the other hand, if a session has not a logical sequence of events, it is a sign that the design has decayed or the developer has lost his tracks. Such a "risky" session (*e.g.,* session A) should be reviewed carefully.
- Since development is incremental, the main concepts of a given feature are defined first. Later, secondary concepts are defined and primary concepts are refined. If a feature implementation is reviewed according to its timeline, a basic version of it can be reviewed first, which is smaller, more concrete and thus more understandable. Only after the general feature is defined, improvements such as optimizations, peculiar cases and generalizations are implemented. This process allows one to understand the general idea of a change first and the details later, just as the developer proceeded. Following the steps of the developer leads to a more natural and progressive understanding of the code. Session B defines first the core of the visitor pattern (B1) – the double-dispatch mechanism and the base class –, and then defines a trivial implementation (B2). A real use case is then built by refactoring the printing report feature to the visitor hierarchy (B3), and finally the address-checking feature is implemented in these term, using both new code and old code from session A (B4). Viewing the implementation in this order makes it easier to understand.
- Individual changes themselves have more context: refactorings and bug fixing can be easily identified. Hence less time can be allocated to understand refactorings, which do not change the behavior of the code, and more to understand bug fixes or design-level changes. In our example, the changes in B3 can be reviewed faster than those in B4.

4 Development Sessions

To create a concise but effective vocabulary when we talk about the different types of sessions, we use a metaphor taken from Brant's "How Buildings Learn" [1], where he describes buildings as multilayered structures where inner layers change faster. Brant's book is about architecture and therefore his layers are (from inner to outer) stuff, space plan, services, skin, structure, and site. The idea is that for example "stuff" (the furniture) is changed more often than the space plan of a house, which is also changed more often than its skin, *etc.* Transposing the metaphor into the domain of software development works, because the bodies of methods are changed more often that the class definitions which are changed more often than the structure of packages, *etc.*

Session	Description
Decoration	The finest level of changes, such as modifying a method body.
Painting	Adding methods to a class.
Restoration	Performing refactorings on methods and classes.
Masonry	Adding one or more classes to the system.
Architecture	Modification to the system's structure, sich as adding packages or large-scale addition of classes.
City Planning	Major overhaul of the system's architecture, unlikely to appear in a single session and therefore not discussed in this paper.

Table 1. Session Types.

Based on this metaphor, we identify a number of session types (defined in Table 1), that we use in the remainder of this paper.

4.1 Session Metrics & Characteristics

In the next section we perform both a quantitative and qualitative evaluation of sessions. To automatically detect the types of the sessions we make use of a wide set of metrics. We present them only briefly in Table 2, since the metrics are not the focus of the paper, but only a means to characterize the sessions. We are interested in detecting sessions with the following characteristics:

Refactoring-based/-less: Sessions that contain actions related to refactorings.

Focused/unfocused: Sessions focusing on a small/large number of methods and classes, indicative of a tight/wide focus.

Adding/modifying: Sessions that consist of additions of new artifacts or only modifications of existing ones.

Metric	Description
SLM	Session Length – expressed in minutes
TNC	Total Number of Changes, *i.e.,* developer-level actions performed during a session
SA	Session Activity, *i.e.,* changes per minute ($SA = \frac{TNC}{SLM}$)
FOCUS	Session Focus, $FOCUS = \frac{NTC+NTM}{TNC}$
NAM	Number of methods added during a session
NCM	Number of methods changed during a session.
UNCM	Unique number of methods changed ($UNCM \leq NCM$).
ACM	Average changes per method ($ACC = \frac{NCC}{UNCC}$).
MCM	Most changed method, the highest number of changes applied to a method during the session.
NMR	Number of method-level refactorings performed during a session.
NAC	Number of classes added during a session
NCC	Number of classes changed during a session.
UNCC	Unique number of classes changed ($UNCC \leq NCC$).
ACC	Average changes per class ($ACC = \frac{NCC}{UNCC}$).
MCC	Most changed class, the highest number of changes applied to a class.
NCR	Number of class-level refactorings performed during a session.
NTC	Number of touched classes, *i.e.,* classes that were modified or added.
NTM	Number of touched methods, *i.e.,* methods that were modified or added.
ANMC	Number of methods changed per class ($ANMCC = \frac{(NAM+NCM)}{NTC}$)
MMCC	The highest number of methods added/modified in the most changed class.

Table 2. Session Metrics.

Large/small: Sessions with a large/small number of changes.

Long/Short: Sessions that last from a number of hours down to a couple of seconds.

Unique/repeated modifications: Sessions that modify artifacts only once vs. sessions that repeatedly change the same artifacts. The latter is a sign of the implementation of complex functionality where the developer takes small steps to modify parts. It may also be that a bug has been discovered in the implementation of a feature and several hypotheses are explored to fix it before finding the right one. In both cases, a method modified several times in a row can be identified as a "risky" method [16], hence its accurate comprehension is critical. The incremental change-based history we recover eases this task.

Sequence of changes: The sequence of changes itself can give insights about the code: A common pattern of change sequence is to first add several entities (classes and methods), and then modify them repeatedly to implement refinements. Another pattern is seemingly random modifications of the system.

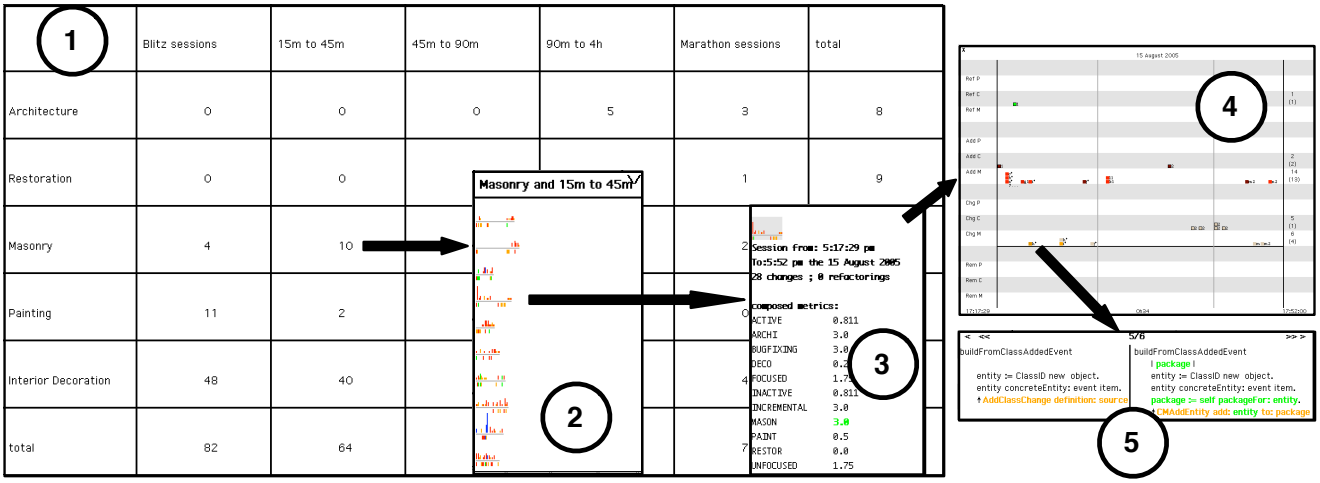


Figure 1. Exploration process

5 Characterizing Development Sessions

We describe the process we used to look for particular sessions in our two case studies (see Figure 1). We use the composite nature of the changes described by our model to adopt a "drill-down" approach to session exploration. We start with (1) high-level characteristics classifying the sessions, then choose a set of characteristics that we want to investigate. This gives us (2) a condensed view of the sessions as "sparklines", a visualization that allows us to choose a particular session to focus on. We can then (3) look at the metrics of the chosen session and from there invoke a (4) session visualizer to have a precise view of the timeline. Individual changes can then be inspected further (5) to look at code-level changes. We describe these steps in detail in the remainder of this section.

We looked at two case studies: (1) Spyware, our prototype, which has been monitoring itself for a period of over 18 months, and (2) Project X, which comes from an external developer monitored over a period of 4 months during his day-to-day programming activities on web applications.

5.1 Quantitative Analysis

Table 3 presents high-level characteristics of the sessions for both projects. Each session is characterized by its type (Architecture, Restoration, Masonry, Painting or Decoration) and its length (0 to 15 minutes, 15 - 45 minutes, 45 - 90 minutes, 90 minutes to 4 hours, more than 4 hours). The session types are not mutually exclusive, *i.e.,* a session can be of more than one type, such as a combined decoration & painting session.

The table reveals that some session types tend to have a characteristic length: Masonry takes more time, Restoration also. Painting on the contrary is an activity which takes less time. There is no typical duration for Decoration sessions, which range from very short to very long ones. We also see that Architecture sessions are rare, while the most frequent type is Decoration. Some sessions that we named *Mixed* cannot be clearly assigned to one of the types, such as sessions in which heavy decoration & painting is going on equally.

Length	0-15m	15-45 m	45-90 m	90m-4h	>4h	Total
Spyware						
Architecture	0	0	0	5	3	8
Masonry	4	10	14	27	2	57
Restoration	0	0	1	7	1	9
Painting	11	2	3	0	0	16
Decoration	48	40	38	55	4	185
Project X						
Architecture	0	0	0	1	0	1
Masonry	0	5	6	8	0	19
Restoration	0	0	0	0	0	0
Painting	3	4	1	2	0	10
Decoration	6	6	4	3	0	19

Table 3. Session Types

We now pick some selected sessions to explain how we drill down from this high-level view to comprehend the sessions in detail.

5.2 Qualitative Analysis

We use the metrics defined in Section 4 to display the sessions in an interactive session table. In all cases, clicking on a cell of the table brings up a sorted list –according to the values of the characteristics of the table– of sessions, visualized as *sparklines*.

The session sparkline is influenced by Tufte's concept

of the same name [1]. A sparkline is a word-sized graphic containing a high density of information, such as:

This figure is an example of a session represented as a sparkline. The gray line in the middle of the figure is a time line. The default resolution of the figure is one pixel per minute. Above and below the time line are bars representing the amount of changes occuring during a given interval (in our case a minute). Above the line are method-related changes: The height of these bars varies with the amount of change performed during the interval. Below the timeline are class-level changes. The class bars do not vary in height as it is rare that two classes are changed in less than one minute. The color of each bar reflects the kind of change happening during the interval. A bar is orange if only modifications happened during that interval. It is red if at least one change is an addition of an entity, blue if one is a removal (superceding red), and green in case of a refactoring (superceding blue).

The sparklines are sorted in a session sparkline list, as shown in step 2 of Figure 1. Sessions can be classified by their characteristics to organize the data and quickly select interesting sessions. The table is interactive: each sparkline can display a text tooltip, and each interval bar can also sum up the changes happening during it in a tooltip. Clicking on a sparkline brings the value of all the metrics of this session (step 3). This gives more context on whether to continue the exploration of this session or not. Session sparklines can also be displayed all at once on the screen, sorted chronologically: This allows us to get a bird's eye view of the entire system evolution, session by session. Viewing all the data in the sparkline session table allows us to quickly identify sessions. A simple click on its sparkline brings up a more detailed view of it, by means of the session visualizer.

5.2.1 The Session Visualizer

The session visualizer (see Figure 2) shows more details about a session, in particular the exact nature of changes performed at a given point in time in a session. Changes of the same type and on the same entities are displayed on the same line, as squares. Change types are: modification, addition, removal and refactorings, while the entities considered are classes, packages and methods. The same colors of the sparkline figure are used. Each of these change figure has an identifier, so that changes applying to the same entities can be quickly identified. They can also display a tooltip summing up the change as text, and upon clicking, a detailed inspection interface is shown. The duration of the session, start date, end date and quantifying statistics are reported on the borders. Clicking on an individual change brings another finer level of details: A view of the entity before and after the change allows to see the sorce code modifications, with syntax highlighting of the changes. This view can be navigated to see the next and previous changes at the entity and session level to get more context if needed.

We now discuss some example sessions. For each session we show the sparkline besides the name and the corresponding session visualizer figure.

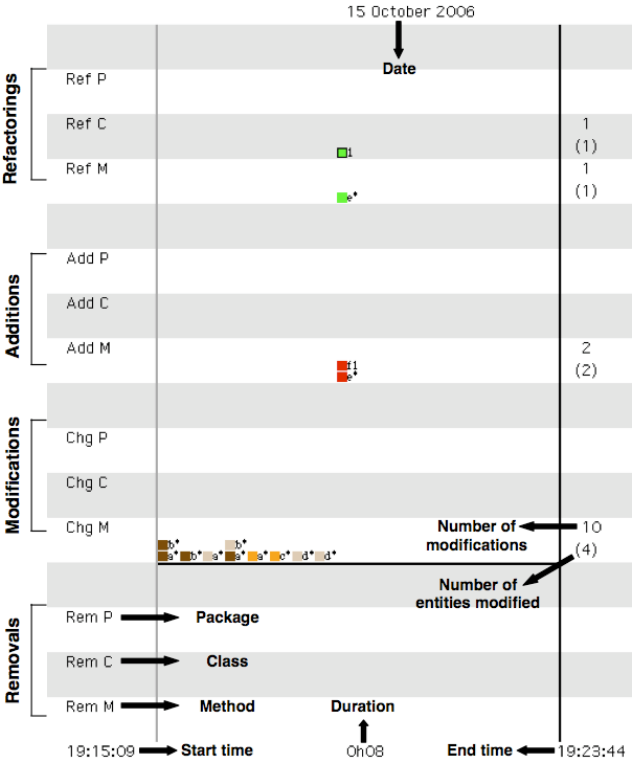


Figure 2. Decoration Session

Decoration Session (Project X)

This short session (8 minutes) consists mainly of decoration, *i.e.,* method modifications. It features towards the end a small amount of masonry and minor restoration. Session C is displayed in Figure 2. Its interesting characteristics are its high activity (1.6 changes per minute) and the first part of it where methods a and b are modified together several times, evoking logical coupling. A look at the source code reveals that they are two HTML generation methods belonging to the same class. Methods c and d, and the methods e and f, are also related to HTML generation.

[1] http://www.sparklines.org

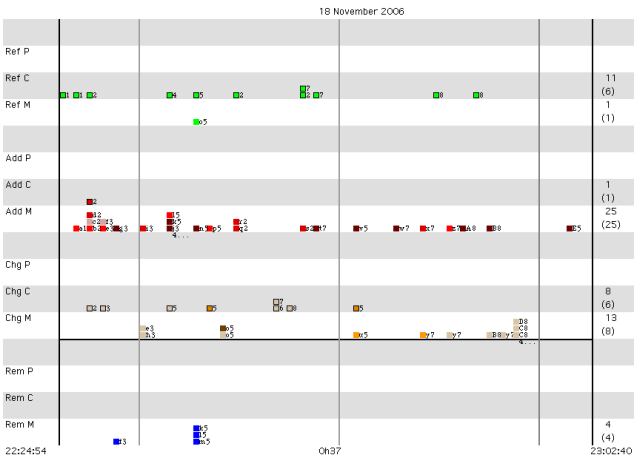


Figure 3. Masonry & Painting Session

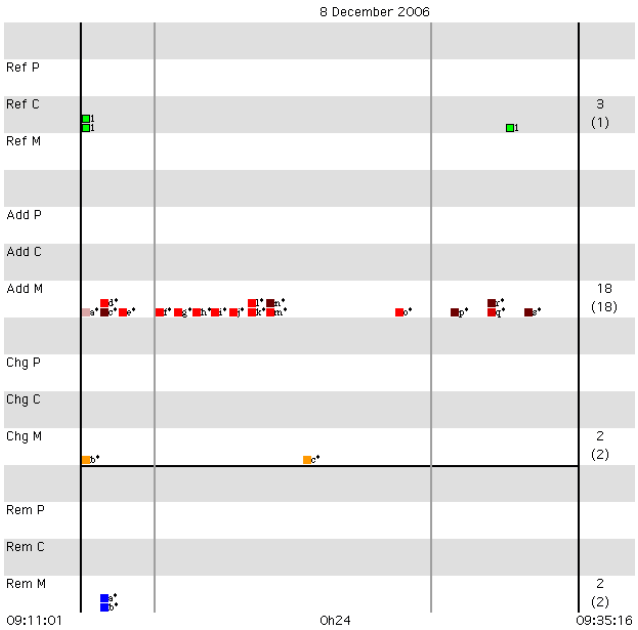


Figure 4. Painting Session

Masonry & Painting Session (Project X)

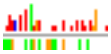
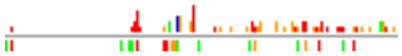


Figure 3 shows an intense 37 minutes long masonry & painting session featuring a lot of class-based development. Indeed, one class is added, which is the focus of the session, while 8 class modifications happen during the first half of the session. Looking at the class modified and referenced in the session, we see that it is included in a hierarchy of classes following the command design pattern [4]. The developer is fast at implementing the new command, which is actually a Composite Command, another design pattern. The methods added to this class show the minimal protocol expected from a member of the command hierarchy: execute, validate and initialize. An extended protocol is added to other classes of the hierarchy with the methods doAnswer and commitToCommands. This session is a good example to follow, should the system need to be extended with a new kind of command by a less experienced developer.

Painting Session (Project X)

Figure 4 shows a peculiar session since its beginning shows the quick addition of methods to several classes. It is again quite short (25 minutes). A closer inspection shows that the methods a, and d, to m have the same name and are added to a hierarchy. They each return a constant, which explains why they are developed in succession. Once this is done, the rythm slows down, and some actual logic is added to the system. This trend is started by method n, which specifies a test that needs to be fulfilled for the implementation to be correct. Later in the session, a strategy for file downloading is implemented relying on two possibilities. It is closely related to the first part of the session since a, d to m were referencing file names, used in methods b and c to build URLs.

Architecture & Restoration Session (Spyware)



We finish with a longer session (see Figure 5) from our own prototype, featuring architectural changes and restoration activities. This session lasts for 2 hours and 25 minutes. During its implementation, the model of SpyWare was extended to include session-level changes, and a simple tool was implemented. From the sparkline we can divide the activity in 3 parts: **F1** shows nearly no sign of activity, **F2** is constituted of two activity spikes stopping at around half of the session. Then **F3** finishes with a more stable output. We see that refactorings are applied consistently during the session, and that **F2** has a higher ratio of additions in its first spike. We now describe each part of the session in detail:

F1: F1 lays the ground work for the session by defining the ChangeExplorer class and changing its sister class, ChangeExplorerTest. A period of perceived inactivity ensues, which can be interpreted as either a design phase or a documentation phase. Since SpyWare was not able at that time to record navigation information, knowing more about the exact activity is not possible.

F2: The first spike adds a new element to the system: an interface centralizing queries to the model and its sister test class. The last methods in the first spike is a stub method called **sessions**, indicating the intention of using the session concept in the ChangeExplorer tool. A short period of inac-

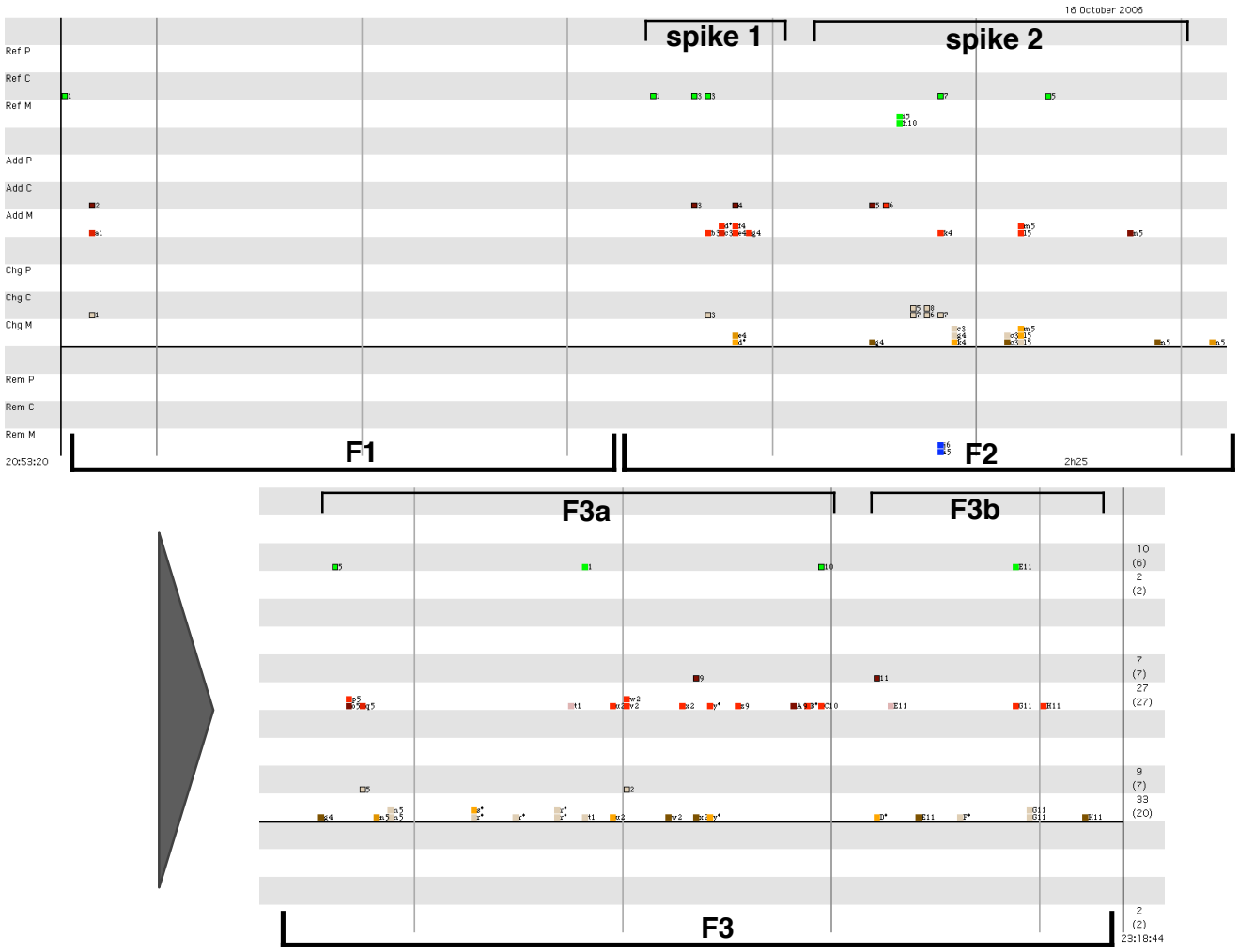


Figure 5. Session F: Architecture and Restoration

tivity follows, quickly replaced by the second spike of **F2**. In it, two classes are defined, the ChangeGroup and the Session class representing a session of changes. Several class modifications are made as ChangeGroup becomes a superclass of several classes. Some methods are pushed up (restoration) and it becomes an abstract class, with Session and Refactoring inheriting from it. Alongside this, the sessions method is modified to exploit these new classes, as well as its test method.

F3: Once **F2** finishes, the architectural phase of the session slows down. The implementation of the actual tool is done in **F3** mainly using Painting. In class number 9 the method **n5**, called changeDescription, is modified repeatedly. A closer analysis shows it returns a textual representation of a change, used in the Explorer tool which is not using graphics. The end of the session adds a new class – 11, called ChangePrinter– and is then exclusively focused on it. Class 11 is in fact used in method D, modified just before 11's introduction. Looking at the code we notice that the class is called AuthoredChange, and the method changeDescription. Looking at the code of the last methods, we deduce that Class 11 is a printing class introduced to handle the changes defined in AuthoredChange, using a double-dispatch mechanism close to the visitor pattern.

To sum up this session, we can discern and describe 5 phases: (1) a design/information gathering phase where development was slow, (2) the definition of the query interaface and the need for sessions, (3) the architectural changes to the model to add sessions, (4) the implementation of the tool, and (5) the implementation of a dedicated printing subclass. Such an incremental vision of the session's history gives a clearer insight on the process than just considering the final outcome: 7 classes were added, 4 other were modified, 27 methods were added and 13 more were modified.

6 Discussion

We have seen by the examples that we have given, that sessions hold indeed useful information to comprehend the development plans of developers, and thus represent a valuable resource for program comprehension.

Sessions are usually spent on parts of the application which are closely tied together, and hence help the understanding of a detailed part of the system. If they are not, it is a sign of lack of focus by the developer, or a possible design flaw in the system. Sessions also follow an incremental logic: functionality is first implemented in a basic way focusing on core concepts, which are then progressively extended. The beginning of a session is thus often a fundamental source of information about the development plans. The elaboration of the concepts in the remainder of a session is also a useful source of information about the way a developer writes code. Versioning systems have no means to store the same amount of information of such granularity–even if a developers commits every minute to the repository, the data recovery process would be arduous and error-prone.

Our approach is the only one to our knowledge able to reconstruct developer sessions accurately enough to ease their understanding. Approaches to program comprehension – assisted by evolution – based on conventional versioning systems can not gather as much information as we can on an evolving system. In particular, they can not *at all* get data on what happened during a development session: They can only store the final outcome of a session. On the other hand, several research tools track events performed by developers in the IDE [17] [15]. These tools could serve as the basis of session comprehension, but to our knowledge there were no attempts at doing this. Although they track user activity, these works do not maintain an accurate enough model of the software to assist understanding of software in an evolution-aware way: They provide links to related entities, but do not show how these entities evolved. The only approach which merges both IDE monitoring and versioning system archives is the one of Parnin *et al.* [11] but it has not been implemented yet, and their later work in [10] only uses IDE interactions.

In contrast, our approach allows us to record a large amount of information about the evolution of a software system by recording IDE events and maintaining an accurate representation of a software system at a domain-specific level, the one of object-oriented languages. Our model is accurate enough to regenerate the source code of any version of the system. This detailed information allows us to model a session accurately as a sequence of changes. We also track other developer actions – navigation in the code, copy and paste actions, debugger usage – although those were not exploited in the context of this paper.

7 Tool Implementation

We have implemented our approach in an IDE plug-in for the Squeak Smalltalk environment, under the moniker "SpyWare" [2] (see Figure 6). We first cover the data retrieval strategy of SpyWare, then briefly list its features.

Data Retrieval. Spyware monitors IDE usage to retrieve semantic changes performed by the developers. Our approach requires the IDE to be open to external contributors by means of a plug-in mechanism [17, 19]. Examples of open IDEs include Squeak[3] and Visualworks[4] for Smalltalk, Eclipse[5] and IntelliJ IDEA[6] for Java.

IDEs maintain an internal representation of the code they manage. They exploit it to offer several language-dependent tools to increase programmer productivity, such as abstracted source code views – beyond the normal source-level, text based view –, refactoring and autocompletion. They also feature event mechanisms which allow the IDE to react to the programmer's activity. We exploit these mechanism to closely monitor a system's evolution, with the following advantages:

- User activity can be processed *as it happens*.
- The time stamps of the changes have an up to the second precision, whereas in a versioning system's repository only the time stamp of the transaction is kept.
- It is possible to be notified of a variety of events to make the analysis more precise. For example refactoring tool usage can be monitored, as well as code navigation or execution errors.

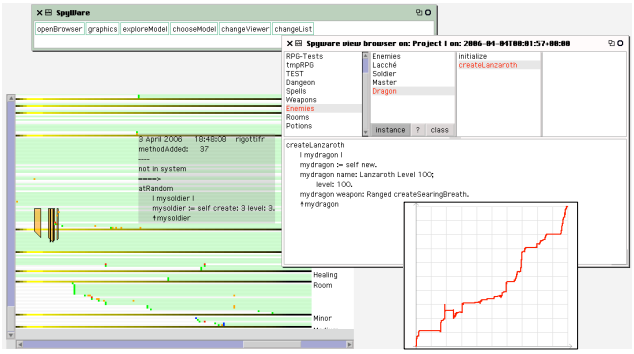


Figure 6. SpyWare's User Interface.

[2] http://romain.robb.es/spyware/
[3] http://www.squeak.org
[4] http://smalltalk.cincom.com
[5] http://www.eclipse.org
[6] http://www.jetbrains.com/idea

Tool functionalities. SpyWare monitors the activity of the IDE user and store code modifications in a change-based repository, alongside other useful information, such as refactorings or navigation paths through the code. In addition to the functionality presented in this paper, SpyWare allows to exploit the change-based information through other interactive data reports and visualizations, such as the change matrix [13]. It can also regenerate the code of a monitored project at any given point in time, and offers dedicated code browsers for this task.

8 Conclusion

We presented an approach to software comprehension at the feature level based on the following 3-step approach: (1) A relevant part of the system (or the whole system) is selected; (2) The selected development sessions are characterized and visually displayed to the user; and (3) Individual sessions can be browsed to support detailed comprehension of the changes belonging to them. By using an accurate model of the *change-based* evolution of a system, we can successfully implement this approach, as shown in the case study.

Our approach fills the gap between versioning-system based approaches, which are not accurate enough to distinguish among individual events happening between developer commits, and purely IDE-based approaches which monitor developer interactions with the IDE, but without a detailed program model. It is the first to our knowledge to permit an accurate session-level comprehension of events.

Our approach can still be enhanced: It requires a heavy set of tools which needs to be ported to other platforms (such as the Eclipse IDE) to be more widely used. We also plan to integrate in our session visualizations the other interactions we have begun to record, such as navigation paths employed by the developer through the system, copy and paste, and errors triggered during execution of the system in the IDE.

References

[1] S. Brand. *How Buildings Learn - What Happens After They're Built*. Penguin Books, 1994.

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

[3] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM'97)*, pages 160–166, Los Alamitos CA, 1997. IEEE Computer Society Press.

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

[5] T. Gîrba, S. Ducasse, and M. Lanza. 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)*, pages 40–49, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[6] C. Görg and P. Weissgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of IWPC (13th International Workshop on Program Comprehension*, pages 205–214. IEEE CS Press, 2005.

[7] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64, 2006.

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

[9] T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings IWPSE2001 (4th International Workshop on Principles of Software Evolution)*, pages 83–86, 2001.

[10] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *ICPC*, pages 13–22, 2006.

[11] C. Parnin, C. Görg, and S. Rugaber. Enriching revision history with interactions. In *MSR*, pages 155–158, 2006.

[12] 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.

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

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

[15] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, sep 2005.

[16] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: raising risk awareness. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 107–110, New York, NY, USA, 2005. ACM Press.

[17] D. Čubranić and G. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, New York NY, 2003. ACM Press.

[18] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[19] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.