Analyzing, Capturing and Taming Software Change[*]

Oscar Nierstrasz, Marcus Denker, Tudor Gîrba and Adrian Lienhard
Software Composition Group
University of Bern
`www.iam.unibe.ch/∼scg`

April 1, 2006

Abstract

Software systems need to continuously change to remain useful. Change appears in several forms and needs to be accommodated at different levels. We propose Changeboxes as a mechanism to encapsulate, manage, analyze and exploit changes to software systems. Our thesis is that only by making change explicit and manipulable can we enable the software developer to manage software change more effectively than is currently possible. Furthermore we argue that we need new insights into assessing the impact of changes and we need to provide new tools and techniques to manage them. We report on the results of some initial prototyping efforts, and we outline a series of research activities that we have started to explore the potential of Changeboxes.

1 Introduction

Complex software systems must change in order to keep pace with changing needs and requirements[20]. If we carry this observation to its logical conclusion, the ability to plan needs to be augmented by the ability to change [4]. Curiously, however, modern programming languages and environments provide little support for the fact that the systems being built will inevitably change [26]. In fact, more emphasis is placed on mechanisms to enforce consistency and to limit the effects of change than on enabling change.

This position paper targets the following questions:

– How can we *encapsulate change* in order to better specify, manipulate and control it?
– How can we *manage the scope of change*, especially in a running system?
– How can we *assess the impact of change* in a complex system?
– How can we *exploit change* to reveal implicit trends and emergent software artifacts?

To answer these questions, we propose to (i) introduce Changeboxes, a programming language construct to package incremental modifications to complex software systems, and use these constructs to express both low-level (syntactic) and high-level (semantic) changes, (ii) develop a *scoped* approach to behavioural and structural *reflection* in which the visibility of reflective features, and thus of changes, can be controlled at a fine level of granularity, (iii) explore techniques for tracing the impact of changes back to their source by monitoring the *flow of object references* in a running system, and (iv) *analyze the evolution* of the software and related artifacts to identify higher-level semantic entities.

[*]Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06), July 2006.

2 Changeboxes: How to encapsulate change?

Present-day development tools and environments do not deal with change in an explicit way. Although integrated development environments such as Eclipse help in integrating different aspects of the development process (*i.e.*, revision control, code refactoring) the current solutions are far from ideal because they are not based on a common infrastructure. They do not support well:

- *Capturing information about changes.* Most revision control systems are only snapshot-based and changes can only be identified *post facto* as differences between versions [23]. Furthermore, they do not provide any semantical information about the changes performed.
- *Concurrent development in a team.* Short synchronization cycles and switching between different versions of a software are hampered by a tedious file- and snapshot-based, update-merge-commit procedure [27].
- *Incremental adaption of a system to cope with a specific change.* Large refactorings can affect many parts of a system and therefore may need a considerable amount of work to be pushed through [8, 17]. While unfinished, the system is not in a consistent shape and cannot be integrated with other work, hindering the evolution or even preventing developers from applying the refactoring.

We propose to investigate means to better support evolution of software systems by explicitly modeling changes at the level of the development environment and programming language. The specific question we want to answer is: *What are the appropriate abstractions to explicitly model the evolution of a software system?*

Classboxes are a scoping mechanism to support unanticipated changes [5, 6]. Classboxes are coarse-grained components that enable developers to locally extend software in a uniform way without impacting other users of that software. As Classboxes have been designed to enable scoping of class extensions, we propose to apply the same principle not only to scope structural changes but also to scope changes over time in an uniform way. From another perspective, as a result of our research in understanding software evolution we concluded that evolution should be modeled as a first class entity [13].

We propose to explore a model for fine-grained changes as first-class entities, and we propose to use this mechanism, called CHANGEBOXES, to express both time-based and structural scoping mechanisms. CHANGEBOXES will allow several versions of the same software artifact to coexist and to be runnable at the same time.

We plan to model changes as ubiquitous events in the evolution of a system, that is, each change brought by a developer will be immediately accessible by any other team member. The difference to changes having global effect is that each change is performed within a specific scope, and the environment allows the developer to switch between these scopes as appropriate. We believe this will achieve a more transparent development process, and will provide a rich model for integrating tools and performing analysis for reverse and re-engineering.

We will start by implementing a Smalltalk prototype of CHANGEBOXES which will express the possibility of add or remove methods. The first prototype will be based on the Classbox implementation. The next step will be to enable the addition or removal of classes, instance variables and changes in inheritance relationships.

Changebox Prototype

We have started our prototype from the approach implemented in Classboxes. In short, this consists of changing the method lookup of Squeak to take into account different versions of methods depending on the execution context. This allows additions and changes of methods to be expressed. We have further extended this approach to deal with removing methods.

The sequence of method changes is reified by modeling method versions as first-class entities. The prototype allows one to select any version of the system and execute or browse the source code of the program in this context. This approach is based on the dynamic lookup in object-oriented systems and on the fact that in Squeak we can reify method execution. In Squeak, however, the lookup of classes is not dynamic (*i.e.*, it is performed at compile time), hence we cannot apply the same approach for expressing several versions of a class definition, for example to take into account instance variable or inheritance modifications.

We are currently investigating ways of modifying the class lookup of the system to be dynamic and to introduce namespaces for managing different versions of the same class. Combined, these techniques would allow for looking up the appropriate class based on the active version of the execution context.

A key validation of Changeboxes will be to express not only low-level modifications to specific versions, but to also express higher-level refactorings which can eventually be applied to different versions of different artifacts. A more important long-term goal is to be able to use Changeboxes as a mechanism to push changes through a system in a controlled way, using scoped reflection.

3 Scoped Reflection: How to manage the scope of change?

Not only source-code, but running systems need to evolve as well. Software systems are typically changed by modifying the source code and recompiling the whole system. This rather old-fashioned model of software development suffers from two outmoded assumptions that are increasingly in conflict with the reality of modern software applications. First is the assumption that the system must be stopped, recompiled and restarted. Many of today's software systems must be up virtually all the time. Second is the assumption that the universe is consistent. Software systems today must cope with the fact that libraries, components and peer systems may be based on incompatible versions of interfaces, protocols and standards. We therefore need mechanisms to enable (i) runtime changes, and (ii) scoped visibility of changes.

Reflection is the ability of a system to change itself at runtime [29, 24, 10]. Smalltalk is a reflective system in which the structure of the system is described by classes and can be changed anytime. Aside from these structural reflective capabilities, however, there is no *meta object protocol* to support fine-grained control of behavior. We have recently implemented Geppetto [28], a dynamic runtime meta object protocol for behavioral reflection based on the design of Reflex [31].

We plan to extend the behavioral reflection framework of Geppetto with the notion of scoped behavioral reflection. This means we want not only to control *what* (spatial) and *when* (temporal) to reflect, but in addition to control the reifications based on the control flow. For example, we want to specify that reifications are active only if the control flow originates in a specified sub-system.

We plan to explore a reflective system that not only scopes behavioral reflection, but in addition provides scoping capabilities for structural reflection. In this track we plan to leverage the work on CHANGEBOXES. We plan to use the behavioural reflection framework as a mechanism to selectively scope the application of CHANGEBOXES dependent on a given context. Conversely, we also plan to use CHANGEBOXES as a mechanism for controlling the availability of the reflective mechanisms themselves. Depending on the current context, then, reflective capabilities may be available, or safely locked away.

In the long term, we plan to explore how the combination of scoped reflection and CHANGEBOXES can enable a more general model of *context-oriented programming* [7], in which the structure and behaviour of software artifacts may change dynamically but in a controlled way depending on the dynamic context.

4 Object Flow Analysis: How to assess the impact of change?

Even small changes can break a system in unexpected ways. To fix a problem the developer has to understand the connection between the location where the problem manifests itself and its cause, the change that introduced the problem.

Dynamic analysis covers a number of techniques for analyzing information gathered while running the program [2, 30]. As object-oriented technology became more wide-spread, it was only natural that procedural analysis techniques were adapted to object-oriented languages. In this context many dynamic analysis techniques focus on only the execution trace as a sequence of message sends [18, 22, 9, 35, 1].

In object-oriented programming the understanding of problems is often complicated due to the *temporal and spatial gap* between the root cause and the effect of errors.

Figure 1 illustrates an example of an execution trace of a program. The order of method activations (squares) is from bottom to top and left to right. While the cause of the bug is introduced at the beginning of the execution, the effect occurs much later on (temporal gap). Moreover, the locations of the cause and the effect are distant in the object space because objects have been passed around during execution (spatial gap).

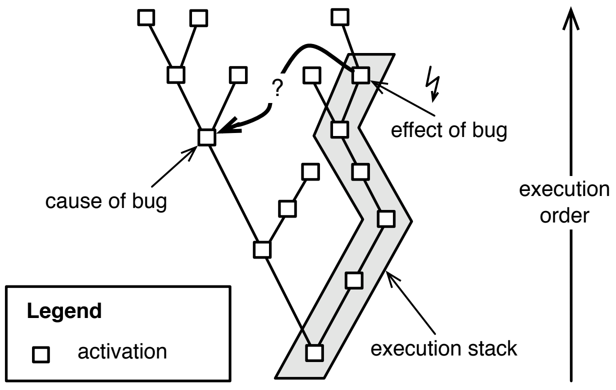

Figure 1: Execution trace emphasizing the cause effect gap of a bug.

Figure 1 also illustrates the execution stack at the point when the error occurred. This is the typical view of a debugger showing the method activation in which the bug is manifested.

The location of the cause of the bug, however, is hidden.

The reason for this gap is static object aliasing – an inherent property of object-orientation. By means of instance variables the flow of an object can bridge the linear sequence of method executions.

Figure 2 illustrates the flow of an object relative to the same program execution. While the object is first passed along with the execution trace, its path later diverges and jumps to distant branches of the tree.

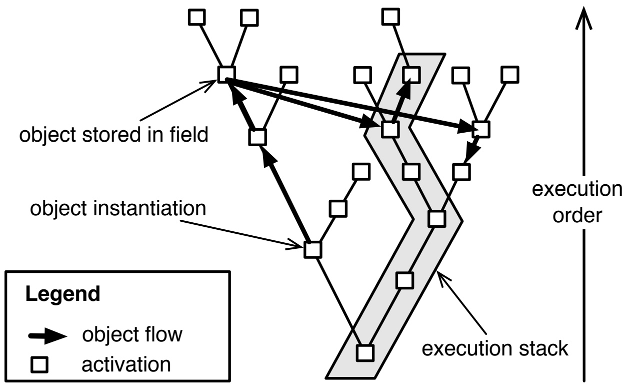

Figure 2: Object flow observed from the execution perspective.

This example illustrates how changes to the software system which modify the behavior of objects may have unexpected effect at distant locations of program execution. Therefore, to connect the cause and the effect of errors we need to trace the flow of objects. This will support the developer in finding errors by allowing him to follow incorrectly behaving objects back along their path.

We plan to investigate a run-time model that complements execution traces with object flow information. We are currently working on a prototype for run-time analysis which introduces the concept of *object alias* to represent explicitly the references to an object.

In comparison to the concept of omniscient debugging [21] which traces the whole execution of the program at a low level, our prototype will additionally provide information about the flow of objects.

Based on this model we plan to build a high-level *object-centric* debugger. By tracing the flow of objects the debugger can provide shortcuts between the chronological sequence of method executions.

Changeboxes will allow for several versions to coexist in the same system. To identify the impact of changes from one version to another, we will implement analyses to compare the dynamic information of the execution of the two versions. In this way we can get closer to identifying the cause of a bug.

5 Evolution Analysis: How to exploit change?

The evolution of software systems is driven by changes at the level of the domain (*e.g.*, features), as change requests are specified in terms of domain concepts. That is why it is important to map domain concepts to code [11, 12, 32].

We propose to pursue several techniques to locate domain concepts from the code and then to analyze how the domain concepts have evolved in the code as opposed to how they are linked conceptually. Different works analyze how the changes in the code relate to the features implemented, by using dynamic analysis [1, 15, 35]. We argue that we need to further investigate this path to recover domain interpretations for the changes in the code, that is, to identify the reasons why the code was changed [16].

Other sources of domain information are the comments and the names of the identifiers [19]. The analyses proposed so far typically focus on one version only. We argue that we need to use information retrieval techniques over several versions of a system to detect when the concepts appear in the system, how they spread and perhaps how they die.

Our main goal is to identify the kind of semantics of changes that would be useful to be encapsulated in Changeboxes. Current versioning systems allow for a textual description to be attached to every new version, but this free-form is difficult to analyze automatically. We plan to allow the developer to relate his changes with the domain concepts using Changeboxes. On the one hand, we want to identify what mechanisms we need for modeling the domain knowledge and relate them to the code changes. On the other hand, we plan to use our automatic analyses to provide hints for the relevant concepts for a change.

From another perspective, software evolution is driven by developers. That is why we need to get more insights into how developers work together to better understand how to support their activities. Several techniques have been proposed to detect patterns of how developers change the system, by analyzing versioning repositories [3, 14, 25, 33, 34]. A particular focus is to combine the developer analysis with the concept location analysis to assign domain concepts to developers. In this way, we link the reasons for change with the different behaviors. By analyzing the developers patterns, we expect to gather requirements for building tools on top of the Changeboxes to facilitate the distributed team development.

6 Concluding remarks

Current programming languages and development environments are focused on limiting the effects of change rather than on enabling change. We have argued here that we need a fresh perspective on how to approach changes, and we have identified several paths to be followed.

On the one hand, we have proposed to tackle changes from a forward engineering point of view. We have proposed Changeboxes, a mechanism that treats changes as first class entities in the environment. Scoped Reflection extends the idea of Changeboxes to cover reflective change. We have proposed to broaden the notion of scope for behavioral reflection and explore the idea of scoping structural reflective change. On the other hand, we have argued that we need to advance our understanding of changes. In this direction, we have proposed a novel approach to assess the impact of change by analyzing how objects flow through a running system. We have also argued for the need to understand how developers drive software evolution and how the evolution of the system relates to the evolution of the domain concepts.

We hope that these efforts will constitute a small step in the direction of freeing programming languages from the stranglehold of static thinking that has historically dominated and (in our opinion) stifled programming language development. Sooner, rather than later, we expect to see a new class of dynamic languages emerge that consider change to be a normal and essential component of the programming language itself.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “A Unified Approach to Composition and Extensibility” (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006).

The research proposed in the paper is planned to be carried out in the context of “Analyzing, capturing and taming software change” (SNF Project Proposal No. 200020-113342, Oct. 2006 - Sept. 2008).

References

[1] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance 2005*. IEEE Computer Society Press, September 2005.

[2] Thomas Ball. The Concept of Dynamic Analysis. In *Proceedings of ESEC/FSE ’99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[3] Timothy Ball and Stephen Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[4] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[5] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[6] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, May 2005.

[7] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming. In *Proceedings of the Dynamic Languages Symposium 2005*, 2005.

[8] Daniel Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of 21st International Conference on Software Maintenance (ICSM 2005)*, pages 389–398, September 2005.

[9] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. Recovering software requirements from system-user interaction traces. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 447–454, 2002.

[10] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA ’89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.

[11] Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution*, 2004.

[12] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99, Los Alamitos CA, November 2003. IEEE Computer Society Press.

[13] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *International Journal on Software Maintenance: Research and Practice (JSME)*, 2006. To appear.

[14] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE)*, pages 113–122. IEEE Computer Society Press, 2005.

[15] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*, pages 314–323. IEEE Computer Society Press, 2005.

[16] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, pages 347–356. IEEE Computer Society Press, September 2005.

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

[18] Michael F. Kleyn and Paul C. Gingrich. GraphTrace — understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88 (International Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 23, pages 191–205. ACM Press, November 1988.

[19] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of Working Conference on Reverse Engineering (WCRE 2005)*, pages 113–122, Los Alamitos CA, November 2005. IEEE Computer Society Press.

[20] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change.* London Academic Press, London, 1985.

[21] Bill Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, October 2003.

[22] D. Licata, C.D. Harris, and S. Krishnamurthi. The feature signatures of evolving programs. *Proceedings Automated Software Engineering*, pages 281–285, 2003.

[23] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch.* Network Theory Ltd., 2003.

[24] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[25] Audris Mockus and Lawrence Votta. Identifying reasons for software change using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 120–130. IEEE Computer Society Press, 2000.

[26] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gaelli, and Roel Wuyts. On the revival of dynamic languages. In Thomas Gschwind and Uwe Aßmann, editors, *Proceedings of Software Composition 2005*, volume 3628, pages 1–13. LNCS 3628, 2005. Invited paper.

[27] Romain Robbes and Michele 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.

[28] David Röthlisberger. Geppetto: Enhancing Smalltalk's reflective capabilities with unanticipated reflection. Masters Thesis, University of Bern, January 2006.

[29] Brian Cantwell Smith. Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA, 1982.

[30] Tarja Systä. Understanding the behavior of Java programs. In *Proceedings WCRE '00, (International Working Conference in Reverse Engineering)*, pages 214–223. IEEE Computer Society Press, November 2000.

[31] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[32] Davor Čubranić and Gail 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.

[33] Jingwei Wu, Richard Holt, and Ahmed Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89, Los Alamitos CA, November 2004. IEEE Computer Society Press.

[34] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 90–99, Los Alamitos CA, November 2004. IEEE Computer Society Press.

[35] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering.* IEEE Computer Society Press, 2005.