Changeboxes

Modeling Change as a First-Class Entity

Masterarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Pascal Zumkehr

2007

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz
Institut für Informatik und angewandte Mathematik

The address of the author:

Pascal Zumkehr
Viktoriarain 6
CH-3013 Bern
zumkehr@students.unibe.ch

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://www.iam.unibe.ch/~scg/

Abstract

Software systems undergo continual change if they want to remain useful over time. However, only limited support for change is offered by current programming languages and development environments. Although various existing efforts try to cope with change and exploit it for different applications, a unifying approach to support software change is missing.

We propose Changeboxes as a generic metamodel to represent change as a first-class entity. Changeboxes encapsulate the semantics of a change process as well as its effects and model the entire change history of a software system. Changeboxes capture changes at the level of the runtime system and the integrated development environment. They are able to record low-level changes as well as complex transformations like refactorings. Each Changebox provides a scope for dynamic execution, whereas several ones can be concurrently active and thus enable different views of the same software artifact within a single running system. Changeboxes can be arbitrarily extended, which enables one to explore several development trails simultaneously. Multiple Changeboxes may be merged in order to combine their changes depending on a customizable conflict resolution strategy.

Our proof-of-concept implementation showed to have an acceptable performance and was used to demonstrate the impact of Changeboxes on various domains. We discuss the advantages that Changeboxes entail for evolving software systems and propose selected topics for further research.

Acknowledgements

I wish to thank my supervisor Adrian Lienhard for his support and guidance during the months this thesis evolved. The numerous discussions helped a lot to sort my mind and to recognize the aimed model. I thank him for the permanent review of my work.

I greatly appreciated the opportunity to write the paper "Encapsulating and Exploiting Change with Changeboxes" (submitted) with Markus Denker, Dr. Tudor Gîrba, Adrian Lienhard, Prof. Dr. Oscar Nierstrasz and Lukas Renggli. The various discussions led to many valuable inputs for this work and my skills in scientific writing could profit highly. A special mention goes to Dr. Tudor Gîrba for the stimulating debates on the Changebox model and for the valuable feedback on this thesis.

I thank Prof. Dr. Oscar Nierstrasz, head of the Software Composition Group, for giving me the opportunity to work in his group and introducing me to the field of software evolution. I especially wish to thank him for revising this master thesis. Many thanks also go to Dr. Alexandre Bergel for his comments and explanations on classboxes.

The year in the student pool would not have passed so fast without the company of Reto Kohlas, Flo Thalmann, Michael Meyer, Rafael Wampfler, Adrian Kuhn, Orla Greevy, Laura Ponisio, Michael Wachter and Jacek Jonczy. Thank you all very much for the help on debugging our router and the Incanto Rondo.

I am deeply grateful to my parents and my sister for all the support and belief they put in me over the past years and for their appreciation for what I do. Last but not least, I would like to express my gratitude to my girlfriend Andreina Badertscher for sharing countless wonderful moments during this year.

Pascal Zumkehr
February 2007

Contents

Chapter 1

Introduction

"Nothing endures but change."

Heraclitus

1.1 Motivation

Software systems undergo permanent change during their life-cycle. Even after the actual development phase, when a system is in productive use, new features and bug-fixes are constantly requested.

Software change occurs in various forms, the most common one being the simple manual edition of program code. But also fully automated, well-defined operations that affect various parts of a system define a change. Generally, we consider a change to be an operation that modifies the *structure* or *behavior* of a software system. The possible semantics being inherent in software change may be manifold. It may denote, for example, bug fixes, adding new features, refactoring or merge operations [Nier04].

While additional features usually can wait for a forthcoming release, changes like bug fixes should be integrated into the running system as soon as possible. Tracking the changes made to different versions of a system and controlling their impact can be a cumbersome task. The modification of running systems is barely possible.

Most programming languages or development environments do not provide any means to cope with change [Nier05]. They imply a closed world view where names for software artifacts like classes, methods or variables are assumed to have a globally consistent meaning in a software system. It is not possible for different versions of the same artifact to be simultaneously active.

To understand the evolution of a software system, its change history and the corresponding semantics are essential [Girb05a]. Conventional programming models are file-based and change is only determinable by comparing different versions of a file. This provides a coarse-grained view without any semantic information, changes that affect the entire system are unseizable. Hence, further development and reengineering efforts have to build on a rather weak foundation.

Change is limited by these aspects of programming systems. By providing only a single view of a system (*i.e.*, the most recent one) without any notion of preceding change processes, a static world is pretended. The existence of change is widely disregarded in this world and very little support for change is offered. This makes it difficult to reason about change, to control its impact or even exploit change in diverse ways.

Problem Statement:

> Most of today's programming languages and development environments provide a closed world view that does not provide explicit support for change.

Various efforts exist to overcome these shortcomings, like versioning tools [Mens02], configuration management systems [Nguy05] or techniques for scoping extensions [Berg05b]. Although they offer solutions to manage change, each of these do so from their own perspective. What is missing is some basic infrastructure that unifies the above concerns and actively supports change.

Research Question:

> How can change be supported and represented in a general-purpose way to meet the demands of evolving software systems?

1.2 Our Approach in a Nutshell

Thesis:

> To actively support software change, we need to recognize change as a primary factor of software development and model it as a first-class entity.

Our solution is embodied in Changeboxes, a general-purpose mechanism for modeling change as a first-class entity.

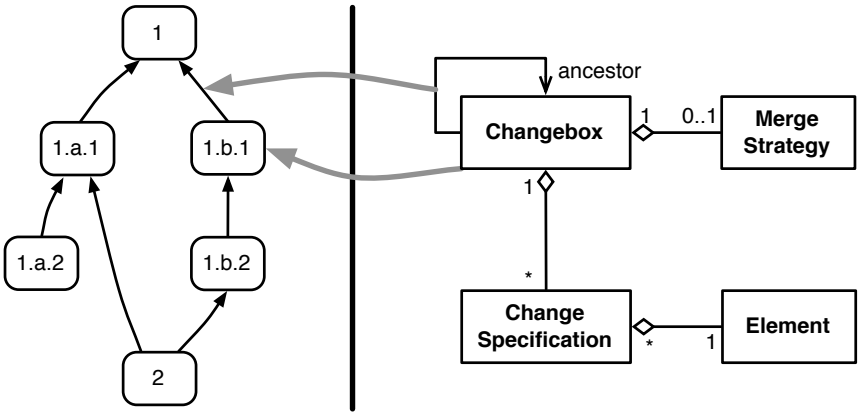


Figure 1.1: The evolution of a system modeled with Changeboxes.

Changeboxes encapsulate single changes to a software system which result in new snapshots of that system. The snapshot a change is based on is referenced as the *ancestor* of the encapsulating Changebox. In the example shown on the left of Figure 1.1, Changebox 1.a.1 encapsulates a change that was performed on the system as defined by Changebox 1. A Changebox represents the structural state of a software system, a new state is attained for every change that is performed.

A Changebox captures both the semantics and the runnable entities that provide the execution behavior of a software system resulting from a change operation. Every change acts on a certain software artifact, an *element*, and can be completely described by a *change specification*. Change specifications hold all the information about a change process that are needed to apply it to a running software system. They can encapsulate both simple changes and more complex refactorings. Change specifications preserve the semantics of change. With this metamodel, shown on the right hand side of Figure 1.1, different kinds of changes can be described in a unified way.

Because runnable entities are directly encapsulated in Changeboxes, it is possible to run a system in different *execution scopes*, which are based on arbitrary Changeboxes. A scope defines the runnable entities that are considered when executing a system. The scope of Changebox 1.a.2, for example, contains all entities modified in 1.a.2, 1.a.1 and 1 and is isolated from the other Changeboxes. Different execution scopes may be concurrently active in a running system.

Our proof-of-concept implementation is written in Squeak [Inga97], an open-source dialect based on Smalltalk-80 [Gold83]. The approach of Changeboxes builds directly on the level of the *runtime system* and the *integrated*

development environment (IDE). Changes to a software system are captured in Changeboxes directly when they occur.

The availability of semantic change information enables fine-grained control over the merge process of several Changeboxes. Many conflicts can be resolved automatically by using the semantic information, especially when refactorings with complex modifications were applied to a system. A merge operation is illustrated in Figure 1.1 where Changebox 2 merges Changeboxes 1.a.1 and 1.b.2. Furthermore, we also offer the possibility to control the semantics of merges, which do not necessarily have to correspond to conventional revision control approaches. For this purpose, customizable *merge strategies* were introduced to control the resolution of conflicts.

Changeboxes can model both *temporal and spatial dimensions* of software change. Changes captured with Changeboxes remain available throughout the entire evolution of a software system. The history can be replayed and analyzed, whereby every single snapshot is runnable. Several Changeboxes denoting an execution scope may be simultaneously active, which makes it possible to support multiple, concurrent views of the same software artifact.

Another rationale of our implementation was the generic applicability of the Changebox metamodel. The design should be open for new, arbitrary elements (acting as change subjects) and any possible change operations working on them. In general, Changeboxes are not tied to encapsulating changes of software artifacts, but they are open to any system with changeable elements.

1.3 Contributions

This thesis presents a novel metamodel for representing software change. It discusses the considerations that led to the design of this model and the accomplished modifications of the Squeak runtime system that allowed us to implement it.

We propose a *metamodel to represent change* in software systems. Only with an appropriate model that covers the semantics of change it is possible to fully exploit and reason about change. Our model encapsulates change processes as well as their effects.

We developed a mechanism that automatically *captures changes on the level of the runtime system.* This asserts that all information about the evolution of a system is made available. Every modification of a software artifact is recorded and encapsulated in a Changebox and is available, including its semantic information, for further analyses. As this happens on the level

of the runtime system, the mechanism is basically independent of where changes originate (IDE, tools, ...) and is always able to capture them.

We provide the ability to create *systems with multiple runnable versions*. While conventional software systems only support one compiled and directly runnable version, every snapshot captured by Changeboxes is runnable without the need for any preceding actions. With the notion of *execution scopes*, different versions of the same software artifact can run concurrently. This opens up new possibilities for many applications, as for example for personalized versions of the same running system.

To facilitate the work of a programmer we integrated Changeboxes into the *development environment*. Various standard tools were adapted to be aware of Changeboxes. Every captured snapshot can be viewed or executed in the appropriate tools, while the opened windows do not all have to provide the same view of the system. A new browser for managing Changeboxes was introduced to act as a central instance when developing within different trails. This allows a programmer to follow his/her conventional development style when using Changeboxes.

1.4 Thesis Outline

Chapter 2 reviews the current state of the field of research on software change. The importance of change and current attempts to cope with it are discussed. We expose the shortcomings of present approaches and identify the basic points to be tackled to overcome them. The advantages of a unifying model for software change are illustrated on a real world scenario.

Chapter 3 proposes Changeboxes as a uniform approach for modeling change and details on the various parts that constitute Changeboxes. Changes are encapsulated as *change specifications* that work on *elements*. Mechanisms to *capture changes* in the system and to provide different active *execution scopes* are described. The rationales behind *merging* several Changeboxes are elucidated and Changebox-aware *development tools* are presented.

Chapter 4 reveals the benchmarks for the execution time complexity of different applications using Changeboxes. We illustrate the ability of Changeboxes to model both spatial and temporal dimensions of change. The former is shown by using Changeboxes to express classboxes, a dynamic module system supporting local rebinding. For the latter, we perform an evolution analysis based on different runnable snapshots of a software project.

Chapter 5 completes this thesis with a discussion about the decisions taken while designing Changeboxes. We mention the issues not covered yet by our model and expose the major contributions of Changeboxes to the field

of software change. To conclude, we suggest future work to advance the Changebox implementation and model.

Appendix A provides a reference for the initial installation of Changeboxes into a Squeak image and contains a short guide to get started with Changebox-aware programming.

Appendix B gives an introduction to Squeak, the Smalltalk dialect used to create our proof-of-concept implementation of Changeboxes. Many essential concepts were feasible in a straight forward way by using Squeak.

Chapter 2

Managing Software Change

Software systems embedded in their operational environment, so-called E-type systems [Lehm80], must permanently be adapted to reflect changing requirements, advancing concepts and improved technology. The acceptance of a system depends on whether it can satisfy the latest needs of its users, otherwise it will no longer be useful over time [Lehm85, Lehm96]. Hence, a system should evolve continuously.

Disregarding this essential requirement, most programming languages and development environments provide a rather static approach that constricts change more than it enables change [Nier04, Nier05, Nier06a]. Systems are regarded as solid entities with a well-specified behavior, which leads to several misconceptions for their development. The problems that arise from these static assumptions are named in Section 2.1.

Various existing attempts address the importance of change and try to cope with it. These technologies provide specific solutions for managing software change, which are discussed in Section 2.2. However, they all act from their own perspective and do not build on a common foundation.

To achieve a unifying model where others could build on we argue to model change as a first-class entity. The fundamental concepts involved in managing software change that such an approach should cover are exposed in Section 2.3.

To illustrate the benefits of first-class changes, we present a motivating scenario in Section 2.4. Various advantages for the development of real world applications emerge from a corresponding model for software change.

2.1 Problems of a Static World

Many programming environments constrict the view of a system to that of a static world where everything is stable and will not change. Objects perform well-defined functions and exhibit a consistent state. Their need to change over time is largely ignored.

The static views of many programming languages prevent change in several cases. Some typical symptoms of this phenomenon include:

Closed-world assumption. Software artifacts, such as types, methods or modules, are generally considered to have a globally valid meaning within a single running system [Nier05]. This meaning is constant and independent of other aspects like a runtime context or the user (be it a human or an object) of such an artifact. The possibility for different, simultaneously active versions of the same software artifact is not given [Smit96, Cost05].

Immutable running systems. Changes usually have to be made at compile time and are only propagated with a redeployment of a system. This often entails a stop and restart which is highly inconvenient for applications that should be permanently running [Sato04, Hick05]. Whenever the form of persistent data changes as well, some ad hoc way to migrate it has to be found by the developer [Nier05].

Backwards compatibility. Whenever an artifact evolves, this must be done in such a way that existing clients are not adversely affected. This leads to an increasing growth of interfaces in many cases [Lehm85]. Although they may be *deprecated*, it can be hard or impossible to definitely remove them [Dig05a, Dig06]. The understandability of a system suffers a lot from this growing complexity.

Reluctance to refactor. Many frameworks provide not only black-box interfaces for passive interaction but also components that can be customized through extension (*i.e.*, subclassing). Refactoring such components may preserve their public interface while breaking implicit contracts visible only to subclasses in client systems [Stey96]. As a consequence, developers can be reluctant to modify framework components that may break client application code [Henk05].

Complex version switching. Systems developed concurrently by multiple programmers require frequent synchronization. This process often includes the handling of file-based snapshots that lack semantic information about the performed changes [Mens02]. Resolving conflicts may involve difficult and unclear decisions. Furthermore, switching between different versions of a system entails a cumbersome update-merge-commit cycle [Robb05] that can be prone to user error.

2.2 Current Attempts to Cope with Change

A wide range of tools and techniques has been developed over the years to cover the evolutionary issues left open by current programming languages. Different attempts address different aspects of software change and help one to manage it appropriately. The following overview reviews various approaches in the field of software evolution. We categorized these attempts into four groups: *language constructs* (Section 2.2.1), *supportive tools* (Section 2.2.2), *merging algorithms* (Section 2.2.3) and *engineering and analysis techniques* (Section 2.2.4).

2.2.1 Language Constructs

The attempts described in this section all operate at the level of the language or the runtime system. By instrumenting certain code constructs, one can achieve a more flexible system in which software artifacts may have different meanings.

Context Aware Systems

Various systems provide multiple different views of their objects depending on the current execution context. They make allowance for the fact that objects in the real world have not only a single perspective, but may exhibit different behavior in certain circumstances [Cost05].

Piccola. Piccola [Ache01b, Ache01a] is a language for specifying applications as compositions of software components. The key mechanism in Piccola is the notion of a first-class namespace (or *form*) which is used to encapsulate the services of a component [Ache00]. Forms also serve as the execution context for scripts. In particular, within a single running application, different execution contexts can be simultaneously active. Piccola does not provide any special support for encapsulating or merging changes.

PIE. PIE [Bobr80, Gold80a, Gold80b, Gold80c] was an experiment to extend the Smalltalk object model with the notion of *views*. PIE is implemented in itself. The source code therefore does not consist of regular Smalltalk classes but rather as PIE *nodes* and provides multiple views of itself. Design decisions are represented from the perspectives of different developers. PIE does not support the possibility of multiple views to be simultaneously active. Before execution, code is flattened to regular Smalltalk classes.

Context-Oriented Programming. ContextL [Cost05] is a language to support Context-Oriented Programming (COP) [Gass98, Keay03]. The language provides a notion of *layers*, which package context-dependent behavioral variations. In practice, the variations consist of method definitions, mixins and *before* and *after* specifications. Layers are dynamically enabled or disabled based on the current execution context. The semantics of layer-composition is fixed. ContextL does not support a notion to express high-level changes like refactorings.

Us. Us [Smit96] is a system based on Self that supports *subjective programming*. Message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. The perspective allows for layer activation similar to ContextL, but does not provide a first-class representation of change.

Aspect-Oriented Programming. Aspect-oriented programming (AOP) [Kicz97, Kend99, Bric05, Char06] provides a general model for modularizing cross cutting concerns. Joinpoints define points in the control flow of a program that trigger the execution of additional cross-cutting code called *advice*. Joinpoints can be defined on the runtime model (*i.e.*, dependent on control flow). Although AOP is used to effect changes on software systems, the focus is on cross-cutting concerns, rather than on software changes in general. The availability of control flow based pointcuts enables different executions to run different code, but this is normally not used to express versioning.

The named approaches allow for context dependent views of objects and respect the ambiguities of software artifacts during their life cycle. Information about how the evolution of these artifacts proceeds in its different contexts, however, is not retained. The semantics of the changes are fixed by the corresponding approaches.

Extendable Modules

With modules, different versions of the same software artifacts may exist separately. Newer module mechanisms allow developers to adapt third-party components in well-defined, separated scopes.

Classboxes. Classboxes [Berg05a, Berg05b, Berg03b] is a module system that provides scoped access to class extensions. A classbox can import definitions from other classboxes and redefine them locally, *i.e.*, these changes are only visible within the local classbox. The classbox in which a class was originally defined remains unaffected and cannot see the changes. Classboxes only support addition and overriding of methods, they do not support removal and thus cannot model general changes. Classboxes also do not offer high-level changes, but only new method definitions. They do not provide any general merging operations. Method extensions simply override existing methods of the same name.

OpenModules. OpenModules [Aldr04, Aldr05] allow clients to adapt a module by providing advice code (derived from AOP) for methods and pointcuts declared in the interface. OpenModules are open for external extension by advice but modular in that they hide implementation information. Advice only works for message sends. Other software artifacts cannot be extended and thus change cannot be modeled in a general way.

The presented techniques allow a developer to extend existing entities in a dedicated scope. Controlling the scope of change helps one to improve the separation of concerns. However, these mechanisms are limited to extending method definitions and do not offer a more general notion of change.

Systems with Multiple Object Versions

Besides viewing or extending existing software artifacts in clearly separated scopes, several mechanisms exist for keeping completely distinct versions of the same artifact in a system. They support objects of the same kind, but with a different shape and mostly serve the purpose to help an existing object evolve.

Gemstone. Gemstone [Penn87, Otis91] provides the concept of class versions. Classes are automatically versioned, but existing instances keep the class (shape and behavior) of their original definition. Instances can be migrated at any time. Gemstone provides (database) transaction semantics,

thus state can be rolled back if the migration fails. Versions of different classes, however, cannot be related with each other to model the evolution of a complete system or to provide separate execution contexts.

Virtual Classes. Virtual classes [Erns99, Mezi03] allow class names to be looked up dynamically. Virtuality of classes, however, is associated with a hierarchy of encapsulating entities, rather than with a particular version of the system as it evolves.

Erlang. In Erlang [Arms96, Arms03] two different versions of the same software artifact can be active at the same time. When code is loaded in the running system, it retains both the old and new version. Calling conventions define which code is called. This allows for a module to continue to execute old code until it is restarted. There are at most two versions active at any time. If a third version is loaded, all processes executing the oldest code are killed. Erlang focuses on providing a robust model for dynamic code loading. It does not try to model change.

Oriol. Oriol [Orio04] has developed an approach to migrate objects in wide-scale applications to new specifications at runtime. The focus is neither on encapsulating change, nor on providing multiple execution contexts within the same running application.

DOORS. DOORS and its Smalltalk prototype [Mezi97] enable dynamic object evolution. Depending on a specific condition, objects can be altered or extended at runtime with *adjustments*. The usefulness of this feature is shown for modeling domain objects, it does not provide a general model of change.

Having multiple object versions in a single system with ways to migrate between them supports the evolution and change of a running system. Entities are basically regarded as modifiable and the change process is actively supported. However, an explicit notion to locate these changes in a certain context is not given by the mentioned mechanisms.

2.2.2 Supportive Tools

Many free and commercial tools capture and store changes performed on software systems. The recorded information may then be exploited in different ways, as the following enumeration illustrates.

Versioning Systems. Versioning systems are used to keep track of all changes made to a software system. They provide a history of the system and allow a developer to explore various branches of its evolution. A wide range of tools with similar functionality exist, such as RCS [Tich85], CVS [Berl90], Subversion [Coll04], Darcs [Roun05] and Monticello [Brya]. These tools all work on snapshots of the source code of a software system. They are text-based and capture changes post facto, *i.e.*, only the results of the changes performed since the last snapshot [MacK03]. Because the snapshot interval mostly depends on the developers, a coarse-grained evolution of a system is represented. But even with automatic mechanisms (*e.g.*, *change sets* in Smalltalk), only the effects without any semantic information of the performed changes are made available.

CatchUp! and RefactoringCrawler. Refactorings change the structure of a software system, but not its behavior [Opdy92, Fowl99]. These changes often affect several parts of a system and may be performed automatically [Dig05a]. With CatchUp! [Henk05], API refactoring operations are captured at the IDE level where they are performed. RefactoringCrawler [Dig05b] extracts the transformations from two snapshots of a system with pattern recognition algorithms. The obtained refactorings from both tools may then be replayed atomically in client code, minimizing manual migration efforts. Replayable refactorings support a developer in maintaining cleaner designs, because client compatibility of refactored versions can be guaranteed. This lowers an important restructuring barrier. General changes (other than refactorings) are neither captured, nor are they related in a way that would allow one to model the evolution of a system.

SpyWare. SpyWare [Robb06] captures software changes at the level of the IDE and represents their semantic information as first-class entity. The captured change operations can be executed to obtain an arbitrary version of the system. The prototype is able to represent changes on packages, classes, methods, variables and statements. Its primary intent is to provide information for reverse engineering and evolution analysis. There is no support for running multiple versions concurrently.

Version Editor. The Version Editor [Atki98] is a history sensitive programming environment that provides online information derived from versioning system repositories. Change information about date, author, previous and related changes may be directly displayed in the editor tool to help the understanding of code. This tool illustrates the importance of evolutionary information for the development of a system, which, however, can only be provided at the granularity available in the versioning system.

The presented tools are primarily concerned with handling software change. They build on top of or independently from a certain language and only take the source, *i.e.*, the non-executable form of the system into account. Appropriate abstractions for software change are defined differently for each tool.

2.2.3 Merging Algorithms

Merging is required to integrate changes from different versions of a system. It is an essential part of versioning systems and other cooperative work environments.

State-Based. Most merging algorithms used for programming languages are text-based and take lines as indivisible units [Hunt76]. Two different modifications of the same line cannot be handled very well, which often results in a conflict the user has to resolve manually. Most popular versioning tools work with state-based merging [Tich85, Berl90, Coll04]. They compare either only the two different snapshots to be merged (two-way merging) or additionally consider the common ancestor (three-way merging) to detect conflicts. Usually, state-based systems do not take semantic information into account [Mens02], neither of the snapshots to merge nor of the changes performed to obtain these snapshots. Conflicts that arise because of semantic reasons might not be detected or not be resolved automatically with pure state-based merging.

Operation-Based. A more sophisticated approach that includes information about actually performed changes is operation-based merging [Feat89, Lipp92, Muns94]. Recorded transformations of a system are partitioned into blocks based on their imposed application order. Whenever the result of applying two transformations of the same block differs depending on the application order (*i.e.*, when the transformations do not commute), they conflict with each other. Directly comparing the change operations of a system may result in a finer-grained detection of conflicts and in the identification of possible application orders that may eliminate certain conflicts [Mens02]. The merged result is more likely to be accurate. For example, when an element is renamed in one branch, references to this element added in the other branch may point to the renamed object in the merged version. The operation-based approach is applicable for arbitrary object types. If required, the behavior of state-based merge algorithms can be emulated as well [Muns94].

2.2.4 Engineering and Analysis Techniques

The need for software change is reflected as well in various (reverse) engineering processes that treat change as a fundamental component:

Software Configuration Management. Software configuration management (SCM) is a software development process responsible for managing changeable entities in a system in order to create different product versions [Tich88, Conr98, Nguy05]. Besides identifying these entities, relationships can be established amongst them and meta-information is attachable. SCM tools offer support to control, audit and report changes on the identified entities and maintain different configuration versions. Versioning systems can be involved for configuration management, which covers not only software, but also hardware, documentation and tests. The target of configuration management is the runnable system.

Model-Driven Engineering. In model-driven engineering, models are considered as first-class entities for the software development process [Kent02]. To reflect changes to a model in different views of a system, *transformations* are executed. These transformations build a crucial part in the evolution of a model-driven system and have been subject to several investigations [Send03, Koeh03, Hear06]. Transformations are expected to work generically on arbitrary models to generate different specific views, for example, to map a business model to a platform specific software design. As transformations work on models, they also hold semantic information about syntactic effects that result from their execution.

Reverse Engineering. Many reverse engineering techniques involve the use of historical data of a software system. The evolving metrics of software artifacts may be visualized to classify evolutionary phenomena [Lanz02, Girb05b]. Information about components that changed at the same time may be used to detect coupling of modules [Gall03]. Yesterday's Weather is used to identify candidates for reverse engineering based on class evolution [Girb04a]. Mining refactoring operations from software archives can provide an insight into the semantic evolution of a system [Gorg05]. The past can provide helpful information to understand and further develop a system. This information mostly origins from versioning repositories and can reconstruct the system evolution only with the found granularity. With more detailed change information, more accurate views would be possible [Robb06].

2.2.5 Summary

All presented concepts and techniques help one to cope with change and have their pros and cons. Programming language constructs allow for modeling systems in which objects may exhibit different behavior depending on the execution context or in which the behavior may be extended by different components. Software artifacts appear changed depending on the perspective. The semantics that are inherent in these changes, however, are not explicitly stated. Such information may be captured by designated tools, possibly even with the complete history of a system. Important shortcomings of these tools are that their change information is not directly runnable and only a selected set of semantics is made available. Merging algorithms help us to integrate changes from different snapshots of a system, while the ones taking semantic information into account can produce more adequate results. Various engineering approaches state different best practices with high-level abstractions but tend to be hard to put into practice due to the missing support by current development environments.

The most important shortcoming is that all these techniques manage change from their own perspective. In each case thoroughly different mechanisms are employed to state and manage change. Some of the presented attempts model the spatial dimension of change by allowing different coexistent versions of the same software artifact, others are more concerned with the evolutionary aspects of change. No common infrastructure allows one to handle change in a unified, elementary way. Without a unifying approach, many resources are spent to cope with change, while much information remains unexploited.

2.3 Modeling Change as a First-Class Entity

Although change is fundamental to software, means to encapsulate and express change in such a way that it can be effectively controlled and exploited are missing. An elementary approach that actively supports change by modeling it as a first-class entity is required.

The following points to represent the temporal and spatial dimensions of change would have to be fulfilled by such an attempt:

- *Encapsulating Change.* Change should be captured as it occurs, *i.e.*, directly in the development environment, or better, at the language level. This asserts that the complete history of a system is available for further exploitation. Not only the syntactic results of change, but also the semantics behind the modification need to be captured so that it is possible to reason about change in a meaningful way.

- *Scoping Change.* Different snapshots of a system may be simultaneously active, even within the same running instance. Changes can be applied to arbitrary snapshots of a system, resulting in new snapshots of that system. A mechanism to control the scope of change is needed, for systems being developed as well as for running systems.
- *Merging and Deploying Change.* Since running systems are subject to modifications as well, means to merge and deploy changes dynamically are required. Different semantics may be inherent in merge operations and have to be facilitated. Modifications should be able to take place at runtime without affecting the availability of a system.

2.4 Motivating Example

To illustrate the benefits of a system that would support the points identified in the previous section, we would like to discuss a few scenarios from web application development. Web applications are a domain in which changes occur frequently and the applications have to be running virtually all the time.

The evolution of a web application along three development branches is shown in Figure 2.1. Each screenshot represents a snapshot of the application that emerged from applying specific changes to its antecedent version. All these snapshots are runnable and encapsulate the performed changes. The arrows point backwards in time to the respective ancestors.

From the initial version 1.0, a branch with a customized version is created and deployed (1). Both versions are simultaneously active on the same running web application container. A developer can safely extend and test the main branch while visitors only see the stable deployed version. Although developing and testing on a productive server might seem highly unusual, numerous advantages may arise. Only one environment has to be set up and potential problems do not have to be reproduced in a different environment in order to fix them. Maintenance tasks are generally sped up. The system supports the separation of different versions.

A second branch from version 1.0 is started for version 1.1, where some changes broke the application (2). Because the broken snapshot runs in its own scope, the 1.0 branch and the deployed version are not affected and operate regularly. Later on, several refactorings (*e.g.*, class renamings that affect many different places in the code) are applied in version 1.1 (3).

After some time, a bug is found in the deployed version. The developers fix it in the 1.0 branch (4) and merge the changes into the deployed branch (5). This change is propagated atomically and directly updates all active visitor

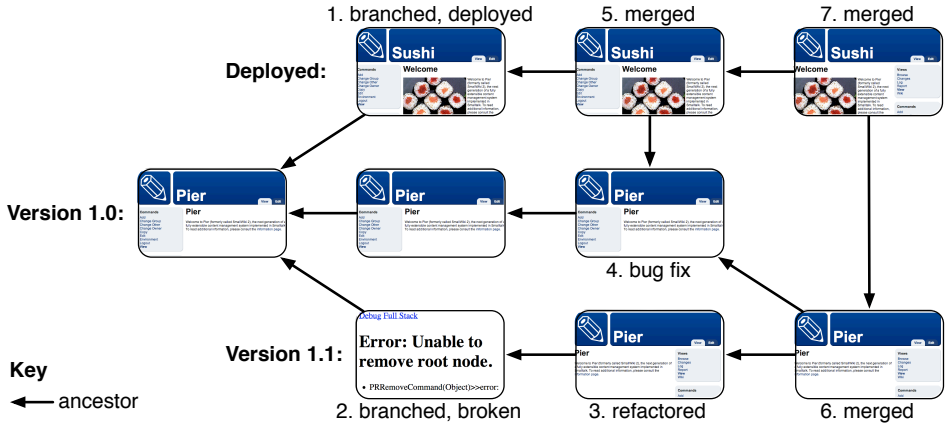


Figure 2.1: The evolution of a web application. Every screenshot represents a running snapshot of the system.

sessions. Because the semantics of the refactorings from the 1.1 branch are available, the bug fix code can be transformed appropriately and be merged into that branch as well (6).

When the stability of the 1.1 branch is assured, the deployed branch can be upgraded (7). Because fine-grained control of the merging process and the semantics of the previously performed changes is available, the precise changes to be merged in can be selected while retaining the customized features of the deployed branch.

The deployed changes of 1.1 have modified the user interface in different ways, therefore this snapshot is not directly activated. Only a selected group is granted access to the upgrade by attaching the corresponding snapshot to the visitor session, which determines the scope within which page requests should be executed. After the approval of the new release all users get to access it. The release is activated for all new sessions, while existing sessions continue with the previous snapshot and then phase out.

A developer who joins the project at a later date will first have to get a basic understanding of the system. The novice cannot only browse through the code of the application, but he may also inspect the changes this code underwent. Information about which parts of the system changed together, at what time and by whom is available. So if a piece of code is completely incomprehensible, maybe a previous version or at least the actual author might provide some hints.

This example illustrates the following advantages for the development of a web application, compared to conventional approaches:

- Several versions of the same running system can coexist without affecting each other. Versions can therefore be customized, developed and tested in the same environment. Problems may be reproduced in the same environment as they occur.
- Changes can be directly integrated into a running system. This may happen immediately, for newly created sessions or only for special users. The currently active snapshot is a simply configurable setting.
- The different snapshots are not only available in textual form like in modern versioning systems, but also in executable form. Tests may be run in different system snapshots to identify the change causing a test to fail.
- Because the semantics of change are encapsulated in every snapshot, refactorings can be captured and applied to different branches by merging the corresponding snapshots. Refactorings are no longer cumbersome modifications that cause vast inconsistencies with related code.
- The meta-information of the changes help one to analyze the system. Fine-grained information is available for the complete history of a project.

Chapter 3

Changeboxes

In this chapter, we present Changeboxes as a unifying mechanism to model software change as a first-class entity. Changeboxes actively support change by *encapsulating their semantics* as well as their effects. They *capture the entire change history* of a system which can be extended at arbitrary versions. Each Changebox provides an *execution scope* for controlling the visibility of changes in a running system, while multiple Changeboxes can be *simultaneously active*. Furthermore, multiple Changeboxes can be combined by *merging*, while a fine-grained control over the merge semantics allows a wide variety of possible applications.

3.1 Overview

A Changebox is a container for a single change applied to a specific version of a system. Each Changebox defines a *snapshot* of a system, upon which new changes can be applied. Normally, changes are performed in a sequential order, hence a Changebox has exactly one *ancestor*. In the special case where several Changeboxes are merged together and changes of different branches are combined, a Changebox can have several ancestors. Over the ancestor(s), all past changes that lead to this Changebox can be retrieved.

Every Changebox defines an *execution scope* for a running software system. This scope is constituted by the complete ancestry of a Changebox (including the Changebox itself), which corresponds to a directed, acyclic graph. The execution scope determines which versions of the classes and methods a running program should use and thus denotes a runnable version of the system. Because every change is encapsulated in Changeboxes, certain scopes might provide *inconsistent views* of a system when the corresponding software artifacts do not work together.

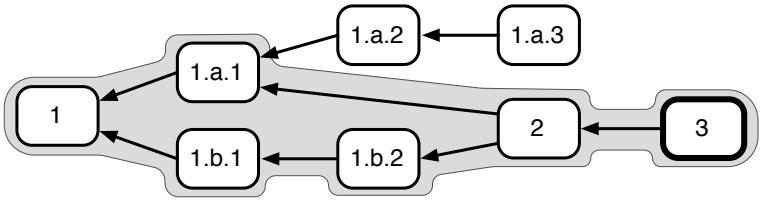


Figure 3.1: The evolution of a system modeled by Changeboxes. The changes visible in the execution scope of Changebox 3 are highlighted.

Figure 3.1 shows the evolution of a system modeled by Changeboxes. Changebox 1 represents the initial snapshot of the system. Changebox 1.a.1 encapsulates a single change applied to Changebox 1, a further change results in Changebox 1.a.2, and so on. These Changeboxes all represent separate snapshots of the system. Changebox 2 performs a merge of the two Changeboxes 1.a.1 and 1.b.2. Changebox 2 is the ancestor of Changebox 3.

The execution scope defined by Changebox 3 (grayed area) is aware of all changes up to this Changebox, *i.e.*, of the initial system 1, of the two merged branches containing 1.a.1, 1.b.1 and 1.b.2 and of the change encapsulated in 3. A separate branch is started from 1.a.1, ending in 1.a.3. These changes are not visible from within the execution scope of Changebox 3.

Once created, a Changebox is *immutable*. A Changebox represents a change applied to the system, which is a past event. To guarantee consistent views of the past, Changeboxes are protected from being manipulated at a later point.

Nevertheless, it is possible to extend a system at arbitrary Changeboxes by applying new changes to them. Subsequent Changeboxes encapsulating the applied changes are created by this process. This is illustrated by Changebox 1.a.2 which represents a new branch originating from 1.a.1. Because every single change applied to a system is represented by a dedicated Changebox, the entire history of a system remains available.

Changebox 2 combines the changes found in the ancestries of 1.a.1 and 1.b.2 by merging them. Hence, this Changebox has two Changeboxes as ancestors. Potential conflicts are resolved by a customizable *merge strategy*, which allows one to define the exact semantics of the merge operation.

For our implementation of Changeboxes in Squeak, we decided to build on the reflective features of Smalltalk rather than attempting to modify the underlying virtual machine. The following issues had to be addressed to make Smalltalk aware of Changeboxes:

- The Changebox *metamodel* needed to be implemented for the software

artifacts and their possible modifications existing in Smalltalk. In Section 3.2, the detailed model for encapsulating change is presented.

- Because Changeboxes offer the possibility to view a system as it existed in the past, changes need to be tracked. The Smalltalk reflective kernel had to be adapted to *capture change* as it occurs. This is discussed together with our approach to record higher level changes in Section 3.3.
- A mechanism that enables *scoped execution* was required in Smalltalk. In Section 3.4, we expose how methods and classes are dispatched based on a defined scope so that the same program may execute in different scopes concurrently.
- *Merging several Changeboxes* proved to be a crucial issue. The steps to keep the merge process open for various customizable conflict resolution strategies that may take change semantics into account are explained in Section 3.5.
- The *tools* constituting the development environment in Squeak had to be made Changebox-aware. They are presented in Section 3.6, together with a utility that is responsible for managing Changeboxes.

3.2 The Changebox Metamodel

A Changebox encapsulates a single change that has been applied to a certain snapshot of a system. To capture the actual change process and its inherent semantics, a Changebox contains one or more *change specifications*. Change specifications identify the subject *element* of a change and define the process for changing an element from one version into another. These classes encapsulated by a Changebox are shown in Figure 3.2.

`Class` and `CompiledMethod` represent the runnable meta-objects that define the structure of a system. These are the most important changeable software artifacts in Smalltalk. Since every change process primarily works on and modifies instances of these classes, we introduce the term *variant* for them in Section 3.2.1. In order to provide an executable system, variants are encapsulated in Changeboxes together with change specifications.

Every change works on the software artifacts that define a system. We call these artifacts *elements*, which can be classes, methods, instance variables etc. The sum of the current versions of all elements constitutes a software system. The means to represent them appropriately are described in Section 3.2.2.

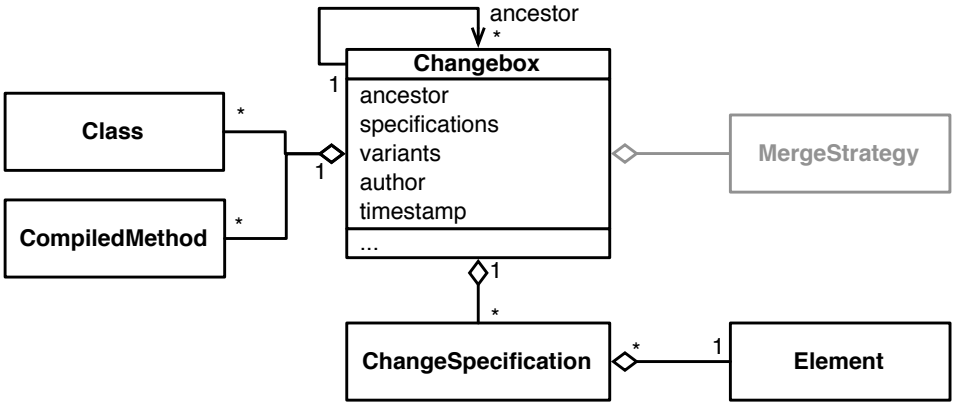


Figure 3.2: Changebox and its associated classes.

A *change specification* holds the modified element and the information about a change operation and can be used to re-apply this change to the system. Thus, a change specification is not bound to a certain snapshot of a system, but only encapsulates a single change process. How change specifications are used to represent the semantics of changes is discussed in Section 3.2.3.

In addition to information about the captured change, a Changebox may contain related data of interest, such as the date and time when the change was applied, the author of the change or any other metadata.

3.2.1 Variants: Instances of Runnable Meta-Objects

When a change is performed in a Smalltalk system, it results in new instances of the runnable meta-objects `Class` and `CompiledMethod`. To ease the formulation when reasoning about these changeable instances, we introduce the term *variant*. A variant denotes a single version of a class or of a method. As Smalltalk is a reflective system implemented in itself, these instances, *i.e.*, classes and methods, are normal objects. They provide all the necessary information for the virtual machine to instantiate new objects (*e.g.*, the class format) and execute methods (*e.g.*, bytecode), respectively. `Class` and `CompiledMethod` are the classes of the most important changeable software artifacts. Their definitions are shown in Figure 3.3.

In a conventional software system, there is exactly one variant for each class and method. Whenever a change is performed, the involved variants are replaced with new ones. Thus, only the last version of a system is runnable. The variants needed to run previous versions are lost. With Changeboxes, every class and method in the system can have multiple variants. For every

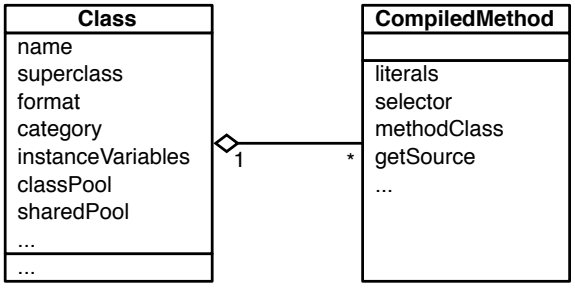


Figure 3.3: `Class` and `CompiledMethod`, the meta-objects modeling a Squeak system.

change that is captured, all the created variants are recorded, along with the semantics given by the change specification in the Changebox that represents the change. Previous variants are not discarded. They are kept in the Changebox that initially captured them. Keeping the variants of all versions of a system in the according Changeboxes offers the possibility to run a system in all versions. Theoretically, all variants could be created on-the-fly from the encapsulated change specifications. This would result in a serious runtime overhead, so variants are cached in the appropriate Changeboxes.

Variants are stored in a dictionary. In the case an element has been deleted, `nil` is put into the dictionary. The representation of removed elements in the variant dictionary of a Changebox is important for various mechanisms explained in the further course.

3.2.2 Elements

In our model, `Class` and `CompiledMethod` are so-called *primitive elements*. Each primitive element is directly represented by a corresponding object (*i.e.*, a variant) in the system. Other changeable software artifacts, such as instance variables or class variables, are so-called *non-primitive elements*. These elements are not represented by a dedicated object in the system, but form parts of a primitive element. For example, field elements are defined by the instance of `Class` they belong to. They are stored in the `Class` attributes `instanceVariables`, `classPool` and `sharedPool`. Changes to non-primitive elements will entail a change of their corresponding primitive element to take effect (*i.e.*, a new variant is created).

To model the different elements found in a software system, a class for every identified element was created as shown in Figure 3.4. Each instance of these `Element` classes represents one specific element, *e.g.*, a certain class, method

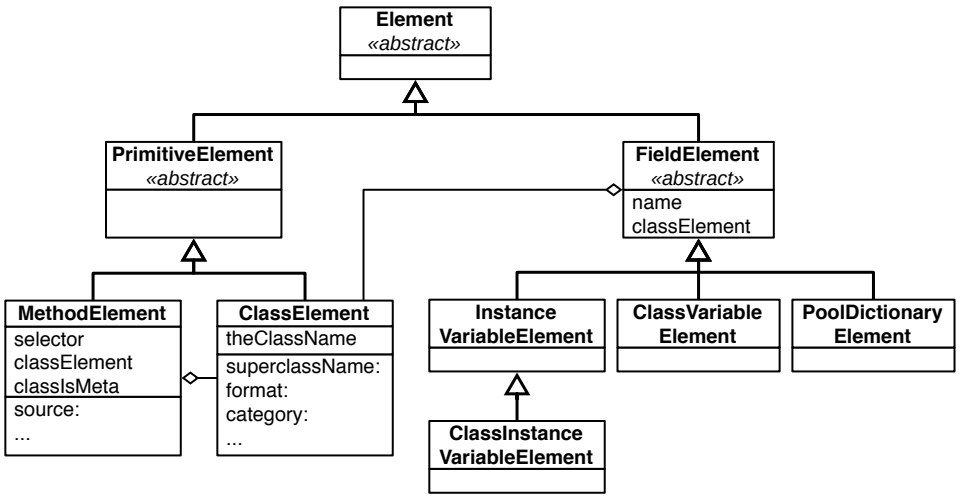


Figure 3.4: All elements with their attributes and changeable properties in Squeak.

or instance variable, independent of the evolution of a system. They provide functionality to access and modify the element. In the following, the model classes are referenced by their common superclass `Element`, the structural software artifacts are continuously named as lower-case 'element'.

An `Element` models the *identity* of a software artifact, but does not provide information about a certain version of that artifact. Each element is mainly distinguished from others of the same type by its name; methods and variables additionally by the class they belong to. Given these attributes, the `Element` object belonging to a certain system artifact can be identified.

Whenever a change affects an attribute that is used to identify an element (*e.g.*, when a method is renamed), the new version of the method will belong to another `Element`. It is the responsibility of the change specification to model these circumstances appropriately.

The most important functions of the `Element` class are illustrated in Figure 3.5 and are discussed in the following.

Each `Element` knows the changeable *properties* and how to extract them from a given instance of their represented software artifact. A property can be the source code of a method or the format of a class. Field elements, on the other hand, do not have any properties in Smalltalk.

Given a change specification, the corresponding `Element` is able to *apply the specification* to the system. This cannot be performed by the change speci-

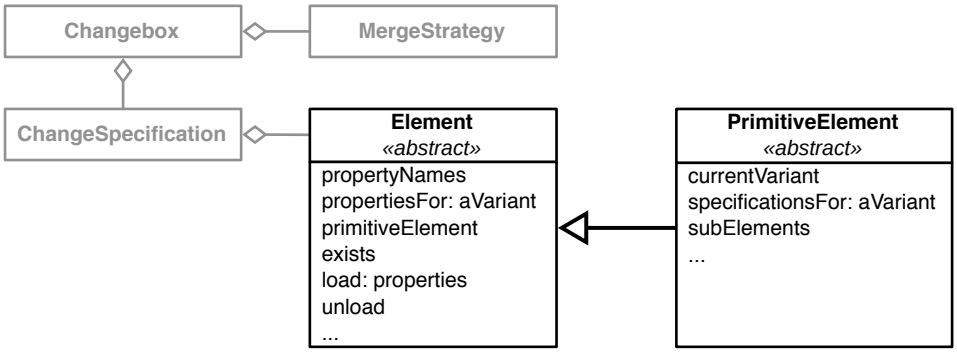


Figure 3.5: The main functionality of `Element` and `PrimitiveElement`.

fication itself as each element is created, modified and removed differently. The corresponding `Element` type has to provide the appropriate behavior by implementing the methods `#load:` and `#unload` for low-level changes and any others required for complex operations.

On the other hand, `Element`s can *generate change specifications* from instances of `Class` and `CompiledMethod` as well, dependent on the current snapshot of a system. Because these instances can contain information about several elements (*i.e.*, also about their non-primitive sub elements), multiple change specifications may be generated. Each primitive `Element` knows the potential non-primitive elements their represented artifacts may contain.

`PrimitiveElement`s are additionally responsible for *accessing the corresponding variants* (*i.e.*, `Class`es and `CompiledMethod`s) in the system, based on the current execution scope. This also includes handling definitions and removals of these elements when the changes should not be captured in Changeboxes, which is required to ensure the functionality of a conventional system.

The different elements identified in Smalltalk and modeled for our proof of concept, as shown in Figure 3.4, are described in the following. This selection is not meant to be complete. Additional elements could be provided for categories, different statements or local variables.

Class Elements

A class is the meta-descriptions of an object, *i.e.*, it defines the object's shape, the associated fields and methods. The name of a class is used as its identifier in the system. In Smalltalk, every class has exactly one superclass from which fields and methods are inherited. As methods and fields are

represented by their own `Element`s, there only remain three modifiable properties that constitute a class element. They are the name of the *superclass*, the *format* of a class[1] and the *category* a class belongs to.

A class is a *primitive element* that has a directly corresponding meta-object in Smalltalk, namely an instance of `Class`. The `ClassElement` provides access to the represented, currently active `Class` instance of the system, as well as to its superclass.

When a change is captured, the `ClassElement` generates change specifications from the newly created `Class` instance. The `ClassElement` generates a change specification for itself and then passes the `Class` instance to all the available `FieldElement`s to create appropriate specifications as well, if necessary.

On the other hand, a `ClassElement` is able to generate a new class from a set of change specifications whose elements (primitive and non-primitive) all contribute to the same class. A specification for a `ClassElement` can be applied on its own as well, defining a new class with different properties as the previous one, but with the same fields. The contained fields depend on the system snapshot the change specification was applied to.

Method Elements

In Squeak, methods are identified uniquely by their name (*i.e.*, their selector), the class they belong to and whether they are instance or class side methods. The *source* code of a method is the only modifiable property[2].

Methods are *primitive elements* as well. In a Smalltalk system, they are represented by the object `CompiledMethod`. The `MethodElement` provides access to the represented `CompiledMethod` in the currently active execution scope, to the `ClassElement` this method belongs to and to the potential indirect super class method in the same scope.

When a method is modified, the `MethodElement` can extract the source code to generate a change specification. As a method does not have any other associated elements, only one specification for each method change is created.

Given a change specification for a method, a `MethodElement` is able to apply

[1] In Squeak, an object can have different formats, defining its shape in memory: `normal`, `bytes`, `variable`, `words` and `weak` [Guzd01].

[2] As mentioned, additional properties and even sub elements such as statements or local variables would be possible. This representation corresponds to our current implementation.

the specification and create a new `CompiledMethod`. This is achieved by compiling the method's source code found in the specification for the class given by the associated `ClassElement`.

Field Elements

A class has several kinds of fields which define different scopes for variables. In Squeak, instance variables, class variables and shared pool dictionaries are possible field types, while instance variables may exist on the class side of an object as well [Guzd01]. A field element is identified by its name and the class it belongs to. There are no additional properties for a field that can be modified, the field either exists in the associated class or not.

Fields are not directly represented with their own meta-object in Squeak, but they are managed by `Class` instances (*i.e.*, they are *non-primitve elements*). Thus, each `FieldElement` belongs to a `ClassElement`. A `FieldElement` can tell whether the represented field exists in the associated class in the current execution scope.

As well as primitive elements, `FieldElement`s are able to create change specifications for their represented field. Such specifications can be applied as well, creating a new `Class` instance based on the preceding one in the current system snapshot. If for example a definition specification for a field is applied, the new class is identical to its predecessor except for the newly added field.

A short definition of the identified field elements in Squeak follows.

Instance Variable Elements. Instance variables store information bound to each single instance of a class. The number of instance variables contributes to the memory footprint of an object. They are accessed using an offset which is compiled into the bytecode representation of a method, thus changing instance variables requires a recompilation of all methods of the affected class and its subclasses.

Class Variable Elements. Class variables are fields that contain information accessible to all objects of the same class. They can be used to share class data between objects. Their name is used to lookup the current value in a class-side dictionary, thus no recompilation of any methods is needed for a class variable modification.

Class Instance Variable Elements. Class instance variables are the same for a class as instance variables are for an object. They are not accessible to the instances of the class, only to the class itself. A recompilation is required for all class-side methods and those of all subclasses when a class instance variable changes.

Shared Pool Dictionary Elements. Shared pools define a scope for variables that are available to all classes and objects, as long as the shared pool field is defined for a given class. They can be considered as global variables and usually contain constants. Pool dictionaries are not widely used in Squeak.

3.2.3 Change Specifications

All of the described elements may be modified in similar ways. Each modification is considered to be a change, which always builds on two basic kinds of changes in Smalltalk: the *definition* and the *removal* of an element. Squeak does not distinguish between the modification of an existing element and the creation of a new one. These two operations are both considered to be a definition change. The two identified simple changes, definition and removal, build a foundation for other, more complex changes. They can be combined in a generic way to create complex modifications, which can still be considered as a single semantic change. For example, pushing up a method to the super class requires the definition of a method with the same content in the super class and the subsequent removal of the original method.

For every kind of change to be captured with Changeboxes, a change specification has to be created to *encapsulate its semantics*. The semantics of a change characterize its intent (*e.g.*, to push up a method), instead of its effect (*e.g.*, create a method with the same body in the super class and delete the original one). The specification is also responsible for storing the information needed to *apply the represented change to the system* (*e.g.*, the method element and the super class to push it up to, but not the actual method content).

For the basic changes (*e.g.*, method definition and removal), the mapping of a low-level change process performed by the runtime system to an according change specification is trivial. However, the system procedures do not always correspond directly to semantic definition of a change. A single action might (i) entail changes to several elements by modifying only one variant or (ii) update multiple variants to reflect the change of one software artifact. Both cases are discussed in the following.

An example illustrating the first situation is class definition in Squeak. A class definition also contains the definitions for the various fields of a class. For this single action, several separate change specifications are generated, namely for the class and for all fields. They encapsulate the fine-grained semantic information of every single definition. To preserve the unity of such a change, all generated specifications are encapsulated in the same Changebox, together with the created variant. In most other cases, nevertheless, a single change specification suffices to represent a change action.

The second case is more complex since other elements may be updated that do not directly belong the the changed element. This is best shown by the definition of an instance variable. When adding an instance variable to a class, in Squeak, the entire class and its subclasses are recreated and all their methods are recompiled in order to reflect the new offsets of the instance variables. However, semantically, the definitions of the classes and methods have not been modified, only their runnable representations needed to be updated. For this reason, only a single change specification for the instance variable is created and encapsulated in the according Changebox, together with all updated variants.

In this way, change specifications encapsulated in a Changebox are independent of the system snapshot they were initially captured in. Applying a definition specification for an instance variable updates the corresponding elements generically, even if they are different in another snapshot. If specifications for the updated classes and methods would be generated as a consequence to an instance variable change, applying these specifications would revert the entire classes to the point an instance variable change was originally captured.

Hence, a change specification does not encapsulate the result, but rather the process of a change. Only in this way, can the semantics of a change

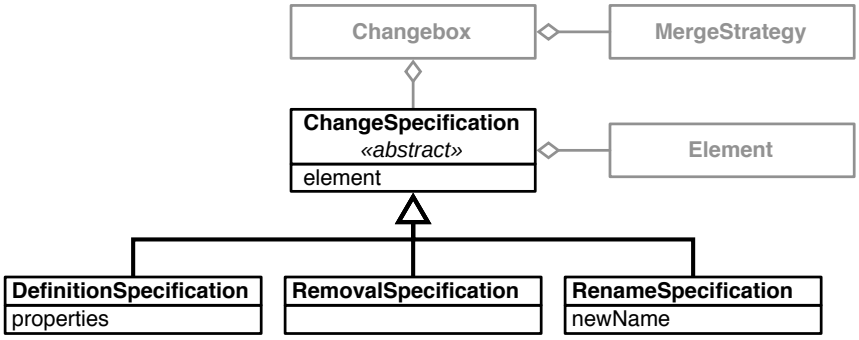


Figure 3.6: The implemented `ChangeSpecification`s and their properties.

be preserved at an appropriate level. The changes represented in our implementation of Changeboxes are shown in Figure 3.6 and are described in the following. Additional specifications could be implemented as long as their represented change is defined for the corresponding elements.

Definition Specification

The most frequently applied change to a software system is the definition or redefinition of an element. The specification for such an operation has to contain all the properties that distinguish one instance of the defined element from another. Given these properties, the corresponding `Element` has to be able to create a new software artifact.

Removal Specification

Whenever anything is removed from a software system, this is represented by a removal specification. This specification only knows the element to remove, no additional data is needed for this purpose. When an element exists in the current system snapshot, it is possible to apply a matching removal specification to discard this element.

Renaming Specification

The renaming of an element can already be considered to be a more complex change, where a copy of the element with a new name is created and the old one is removed. Instead of modeling this with a definition and a removal specification, which would affect the semantics of the change, a new specification type was introduced. The created copy of an element should be identical to the previous one except for the name. Therefore, a renaming specification does not contain any other information than the element in question and its new name. `Element`s that support renaming have to implement a `#rename:` method. A renaming specification can be applied to any definition of an element, preserving its other properties. Named references to the corresponding element might be updated automatically when a renaming specification is applied.

3.3 Capturing Changes

Having defined a metamodel to represent changes of software artifacts, a mechanism to capture these changes and instantiate the encapsulating Changeboxes is required.

Changes are usually performed incrementally by a developer. To keep track of the most recent Changebox to which further changes can be applied, the concept of a *work session* is presented in Section 3.3.1.

Two different ways of applying changes have been identified in Smalltalk. Basic modifications of primitive elements (*i.e.*, the compilation of methods and the building of classes) are performed using the system's reflective libraries. To capture these processes with Changeboxes, the according mechanisms have been intercepted as described in Section 3.3.2.

More complex changes that may involve the modification of several elements are usually performed by appropriate tools of the development environment. Section 3.3.3 suggests to record such changes at the point of their origin to capture the complete semantics.

3.3.1 Work Sessions

With the Changebox model, every change is performed with respect to a certain snapshot of a system, which is represented by a dedicated Changebox. As an interface to external tools, we introduce the concept of a *work session*, which keeps track of the most recent Changebox and thus allows one to apply changes sequentially. A work session starts from a Changebox and remembers the most current one when additional changes are performed. Like a Changebox, a work session defines an execution scope, which is detailed in Section 3.4.

Work sessions are responsible for recording all changes that happen to a system. When a change is captured, a new Changebox is created with the one referenced by the current work session as its ancestor. The work session then updates its reference to the new Changebox. Like this, a work session captures a sequence of changes and applies them in one line. In Figure 3.7, the shown work session manages the middle development line A.

Normally, a developer is working within one work session. Whenever the

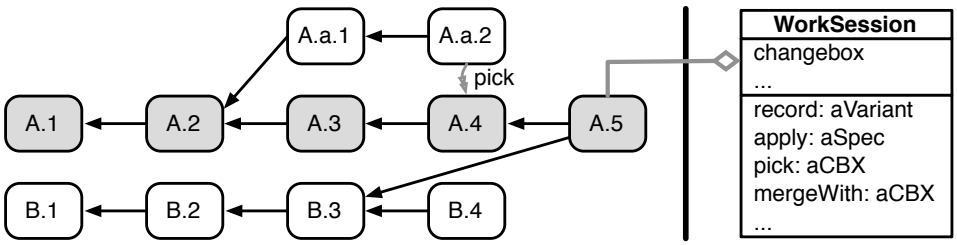


Figure 3.7: Three work sessions illustrating branching, merging and picking.

developer wants to explore a second approach, it is possible to create a new work session, *i.e.*, a *branch*, with an arbitrary Changebox as its initial snapshot. This is illustrated by the upper line which started from work session A at A.2 and continued with A.a.1 and A.a.2. The changes appended there are completely independent of the changes in other work sessions. Instead of undoing changes, such as for example removing previously defined elements, a developer can just go back to the desired Changebox and continue a different work session from there.

Work sessions cannot only be split up into different ones when creating a branch, they are also responsible for *merging* other Changeboxes with the most recent one. The changes from other branches are then visible in the current work session as well. Changebox A.5 merges B.3 of work session B into A, making the changes from B.1 to B.3 available in A. Virtually any set of Changeboxes can be merged together, combining the different classes and methods found in their respective ancestries. Potential conflicts between elements have to be resolved during the merge process, which is encapsulated in a new Changebox. The details of merging can be found in Section 3.5.

Instead of merging complete branches, it is possible to pick some Changeboxes from one work session and apply the contained change specifications to another session. This process is known as *cherry picking* and may be used to apply bug fixing software patches in different development branches. As only the specifications are picked, new Changeboxes are created in the active work session to represent the applied changes. In Figure 3.7, the specifications of A.a.2 are applied to the snapshot defined by A.3, which results in Changebox A.4. The changes from A.a.1 are not integrated into A. Currently, the semantics of a cherry picking operation are not explicitly represented in the Changebox model besides the results of applying the corresponding change specifications.

3.3.2 Capturing Simple Changes

Changes in a Smalltalk system, *i.e.*, the compilation of methods and building of classes, are performed through the reflective kernel. Previously existing variants are discarded and replaced with the new ones.

Squeak provides a notification system for changes of methods and classes. Registered objects get notified on class and method changes. Because these notifications only provide a read-only access to the changed elements, there is no possibility to take action in the change process. With Changeboxes, the semantics of defining and removing elements changes slightly, which makes it necessary to alter the modification processes.

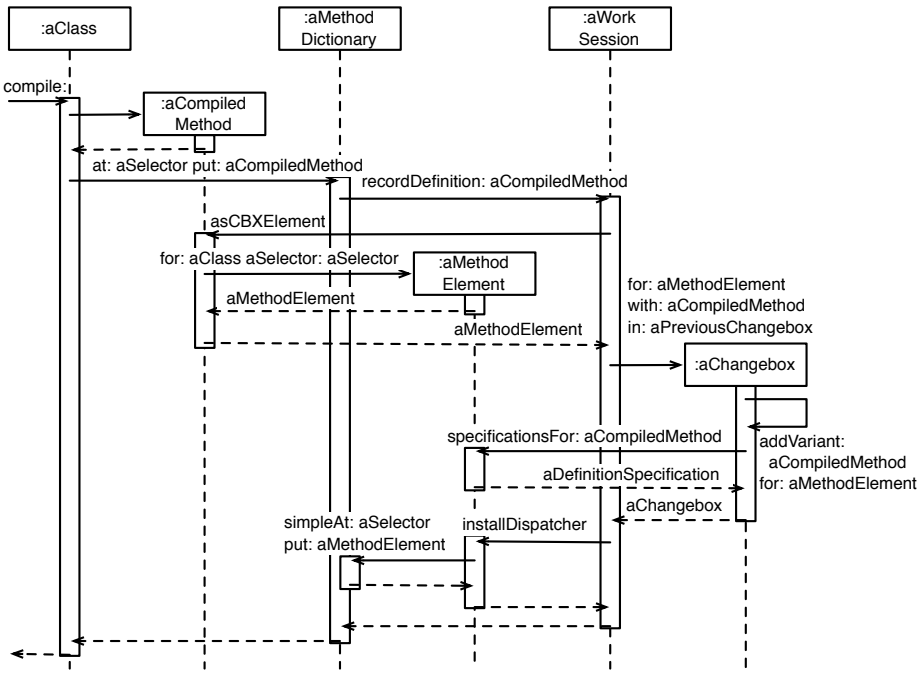


Figure 3.8: Capturing changes bottom-up: UML sequence diagram of a method creation.

A change should only affect the snapshot of a system defined by the current Changebox, other snapshots should not notice anything of that change. In a conventional Smalltalk system, changes to certain elements can result in further actions to adapt the system. For example, redefining a class causes all pointers to the old class variant to be changed to the new one in the entire system. Removing a class causes the old variant to be added to the collection of obsolete classes, or removing a method causes the compiled method to be discarded from the class' method dictionary. This behavior is not desired for Changeboxes, because it would affect the entire system and all its different snapshots. As a consequence, the modification procedures for classes and methods have to be adapted to the needs of Changeboxes. Simply subscribing to the read-only notification system of Squeak would not have offered these possibilities.

The mentioned side effects in the compilation and class building libraries were removed. Instead of replacing an old variant with a new one in the case of a definition, the system procedures pass every new variant to the currently active work session. This creates a new Changebox containing the recorded variant to represent the change. The diagram in Figure 3.8 shows the creation of a method and all subsequent actions to capture the change

in a Changebox. Corresponding actions are taken to capture the removal of an element.

A recorded variant is analyzed by its corresponding `Element` for the changeable properties. The properties are then compared to those of the preceding specification for the same element, as found in the ancestors of the newly created Changebox. When these properties differ, a change specification with the new properties is created and added to the Changebox.

If the properties remain identical, the element did not change and no specification is created. This may happen, *e.g.*, for method recompilations resulting from an instance variable definition. Because the properties of these methods do not change, no change specifications are generated. Only the recompiled variants are put into the Changebox representing that change in order to have a runnable system.

To access a modified element further on, a dispatcher is installed at the element's original place in the system. How this dispatcher retrieves the correct variant for a given execution scope is described in Section 3.4. In the method creation example from Figure 3.8, the dispatcher is installed in the `MethodDictionary` of the corresponding class.

This *bottom-up* approach allows one to capture every change applied to a system. Every definition, modification or removal of a class or method is recorded by the active work session. A fine-grained capturing of structural changes independent of their higher level intent is made possible by this approach.

3.3.3 Recording Refactorings

The bottom-up approach described above captures every single change of a system separately and generates corresponding Changeboxes. It works well for simple changes like definition and removal, but makes it difficult to record high-level changes that are composed of several individual, simple changes. Refactorings generally belong to this group.

As an illustrating example, renaming a class creates a complete copy of the class with the new name and then removes the old class. With the current approach, several definition specifications would be created because all copied elements (*i.e.*, the class, its fields and methods) are assigned to the new class name and thus have their identity changed. Because the copied elements are recorded sequentially, the specifications would all be captured in separate Changeboxes. Our requirement to encapsulate a single change in one Changebox is not met this way. Furthermore, the created change specifications would present semantic information that is not appropriate. Even

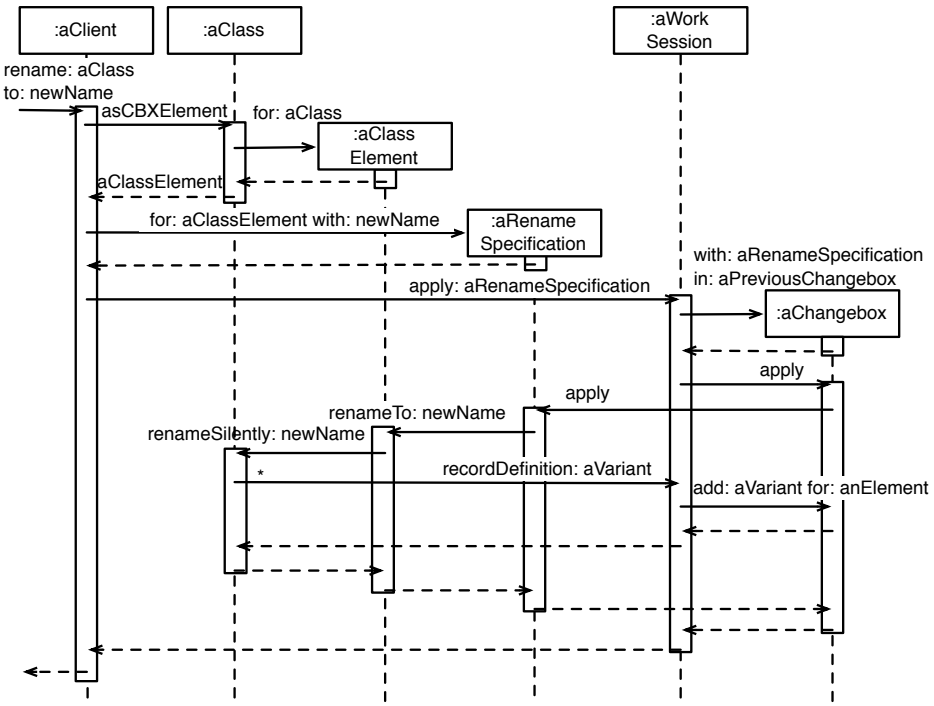


Figure 3.9: Capturing changes top-down: UML sequence diagram of a class renaming.

though the methods of the renamed class are not modified, corresponding definition specifications are created for every method present at the point of the renaming. This might distort the results of applying the captured renaming process to different system snapshots, where the same class may contain other methods, which would be overridden.

To capture the semantics of such a change as precisely as possible, a single specification for the renaming process has to be created. The updated method and class variants should all be captured within a single Changebox to preserve the unity of the change.

Generating the appropriate specification for a refactoring from the variants captured by the bottom-up approach would be ambiguous and error-prone, if not impossible. One common property of refactorings is that they do no change behavior, but only structure [Opdy92]. Hence, a refactoring does not need a lot of information to be specified and works generically on the present elements, which may be different in every snapshot of a system.

As a consequence, we introduce a *top-down* approach for recording refactorings. The exact process is illustrated in Figure 3.9. First, a specification

for the refactoring is created, which is straight forward. This task can be completed by an IDE tool that is aware of the data to perform a refactoring. For a renaming, for example, an instance of `RenameSpecification` is created by specifying the desired element and its new name. In a second step, the specification is applied in the active work session, which creates a new Changebox for this change. All variants produced by applying the change specification are captured in the new Changebox, but no additional specifications are generated for them. Because a change specification encapsulates the process of a change, the result may look different, depending on the system snapshot the specification was applied to.

This approach for recording refactorings works for simple changes as well. In contrast to the bottom-up approach, change specifications have to be created by an external source. The advantage of the bottom-up approach is that it is able to capture any change independent of its origin. Even changes initiated by Changebox-unaware tools can be captured in this way. Furthermore, in the case of a simple definition, determining the property differences to the preceding specification might be tedious for an external tool.

For these reasons, both approaches are applied to record changes. Simple changes can be captured by the bottom-up approach. Advanced refactorings that are easy to specify make use of the top-down approach. Additional refactoring packages which are not part of the Squeak kernel may be consulted to apply more complex change specifications. In this way, the advantages of both approaches find their appropriate applications.

3.4 Scoping Execution

A conventional Smalltalk system does not offer the notion of execution scopes. Only the most recent versions of the different software artifacts are present in a system.

The representation of an execution scope should provide the following facilities (also visible in Figure 3.10). It has to be possible (i) to *lookup the active variant* of a given element and (ii) to *record changes* performed in this execution scope. The lookup for the current variant has to be unambiguous, *i.e.*, only one variant for each element can be active in a given scope. Hence, a scope fulfills the *flattening property* [Nier06b]. Point (ii) is optional, since a scope could be immutable as well.

Both Changeboxes and work sessions may define an execution scope. A Changebox encapsulates `Class`es and `CompiledMethod`s which may be recursively looked up over the ancestors. A work session is responsible for

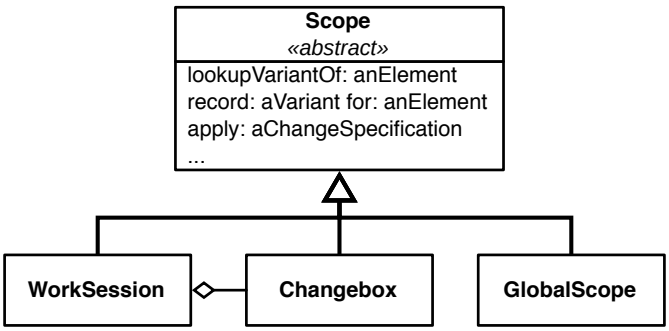


Figure 3.10: The abstract `Scope` class and its subclasses.

capturing changes and can delegate a lookup to the referenced Changebox. Since this reference is incrementally updated, the changes performed in a work session become automatically visible. Changeboxes alone are immutable, so potential changes are discarded when a Changebox is set as execution scope for applying changes. In this way a sandbox system could be modeled where no changes are possible.

The conventional Smalltalk system libraries are currently not encapsulated in Changeboxes. To represent these globally available parts, we introduced a `GlobalScope` object. Every Changebox has the single instance[3] of this class as its ultimate ancestor. The global scope provides the scope facilities when no Changebox or work session is specified, *i.e.*, it acts as an interface to the modified reflective kernel functions and redirects change requests to the conventional Smalltalk procedures.

As Smalltalk is a completely reflective system that is permanently running, scope only has to be set for runtime. There is no difference between a compile time scope where changes are performed and a runtime scope needed for program execution. How the current scope object for execution can be specified is explained in Section 3.4.1

To enable different dynamic views of a system, it should be runnable in arbitrary versions simultaneously. To control the visible variants when executing an application in a given scope, primitive elements are dispatched using the lookup facility of Changeboxes. The rationales behind this variant lookup for a given scope are discussed in Section 3.4.2. How dispatching was achieved for methods is presented in Section 3.4.3, while in Section 3.4.4 the more intricate solution for classes is described.

[3]Following the well-known *Singleton* pattern [Gamm95]

3.4.1 Specifying the Execution Scope

In order to run code in a given scope, this scope has to be specified somehow. It should be possible to have several scopes active simultaneously in a running system, but for every executed method the scope has to be clearly defined.

Given the possibility of executing statements within a block closure in Smalltalk, we implemented a mechanism to specify a scope for the execution of an arbitrary block of code. This mechanism asserts that only one scope is active for a given message send. The syntax for accessing the current scope object is straightforward. A block closure can be run with a specific scope as follows:

```
CurrentScope use: aScope during: [...]
```

Inside the given block closure, the scope can be obtained by sending

```
CurrentScope value
```

The specified value will be returned and can be used in the block. If no value is explicitly specified, *i.e.*, the block is run without a `#use:during:` statement anywhere in the method execution context stack, the instance of `GlobalScope` is returned when calling `#value`. This enables a unified handling as the returned object is always of type `Scope`.

Storing the Scope

We investigated several mechanisms for storing the current scope object and found an additional instance variable in `Process` to be the most efficient. In Squeak, a `Process` object represents a running thread. The instance variable added to `Process` can hold an arbitrary object for every process and is accessed by `CurrentScope`. In this way the scope can be set for a certain block of code running within one process.

It is essential that the scope variable is set back to the previous value when the executed block exits. This is required because processes may span over several tasks, only few of which might actually be run in a special scope. As an example we mention the GUI process that renders the different windows in the Squeak world, whereby different class browser windows can represent the system in different scopes.

A great advantage is that the same block can be run in different scopes at the same time, *i.e.*, in different processes. There is no need to adapt a block to the scope it should be executed in.

Discontinued Alternative: Dynamic Variables

Another possibility we considered was the use of dynamic variables. They build on `Notification`s, which are part of Squeak's exception handling API. A request for the current scope object would have triggered the notification, which walks up the execution stack of the method from where the request originated until it is handled. The handler for the request then returns the scope object specified earlier and resumes the action at the point where the request has started. This is the same mechanism that exceptions use, except that they usually do not resume the action.

This approach was followed for some time during the development of Changeboxes. During benchmarking, it proved to be significantly slower than a `Process` instance variable, as can be read in Section 4.1.2. Dynamic variables did not reveal any special advantages over the `Process` instance variable, so their use was discontinued. In certain situations, a `Process` instance variable is even more accurate. When forking processes, the content of the scope instance variable can easily be copied to the new process, while dynamic variables would have required the duplication of the complete method execution stack up to the fork message in order to retain the scope information in the new process.

3.4.2 The Changebox Lookup Mechanism

As already described in Section 3.2.1, all variants resulting from change processes are captured in corresponding Changeboxes. To obtain the runnable variant for a primitive element in a given execution scope, a lookup over the Changeboxes in that scope has to be performed. Remember that a captured variant can also be `nil`, denoting a removed element. Since variants are represented by `PrimitiveElement`s, these objects are responsible to retrieve the active variant for the current scope and to invoke the lookup mechanism:

1. The lookup starts at the Changebox defining the current scope as found in the `Process` instance variable.
2. If this Changebox contains a variant for the demanded element, this variant is returned.
3. If the Changebox does not contain the demanded variant, the lookup continues in the ancestor of the Changebox.
4. If the ancestor is the global scope, *i.e.*, if the end of the ancestry is reached, then the variant from the global scope is returned (which might be `nil` as well).

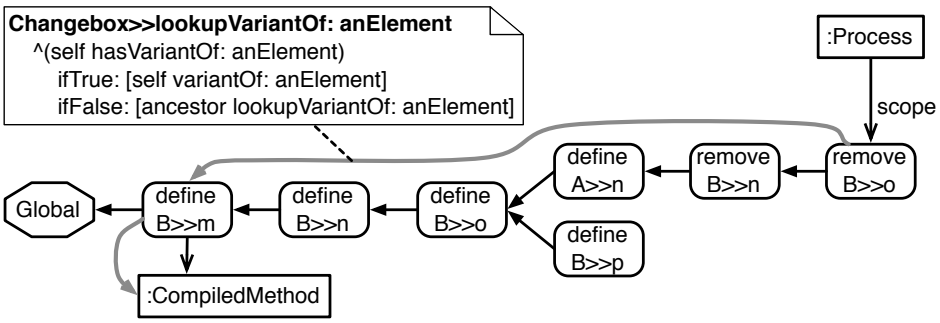


Figure 3.11: Variant lookup over several Changeboxes.

5. If the ancestor is a conventional Changebox, the lookup proceeds recursively with step 2.

The steps followed by an example lookup are illustrated in Figure 3.11. The method `B>>m` should be retrieved. No variant is found in the first Changebox, therefore the lookup continues in its ancestor. Only in the last Changebox, a variant for this method is found and can be returned. A lookup for `B>>n` would find a variant in the second Changebox. Because the method was removed there, the returned variant is `nil` and the previous definition of `B#n` is overridden. When retrieving `B>>p`, the algorithm does not encounter any variants up to the global scope, and `nil` is returned as well. Because `B>>p` is defined in a different branch, it is not visible in the current execution scope.

Only one variant per element will be found by this algorithm, namely the latest one created in a given scope. The *flattening property* is observed. The variants in previous Changeboxes are not considered anymore, but they might be the active ones when a system is running in a different scope.

The existence of a variant in a certain Changebox does not have to depend on the presence of a change specification for the same element. As discussed in Section 3.2.3, changes may update variants without affecting the semantics of the corresponding elements. The variants captured from such changes are required to run the system and are considered equally by the lookup algorithm.

If `nil` is found for the element looked up in the variant dictionary of a Changebox, then this value is returned. Whenever the lookup results in a `nil` value, the requested element has been removed and no longer exists in this execution scope.

A big advantage of modeling the global scope with its own object that precedes every Changebox ancestry is the reduced runtime complexity of

the lookup. With this design, every Changebox has an ancestor object with a `#lookupVariantOf:` method. Therefore, this message can be sent directly without testing any conditions. The receiver (*i.e.*, a `Changebox` or `GlobalScope` object) then performs the appropriate actions. This behavior is generally described as the *Null Object pattern* [Wool96].

Caching Lookup Results

Iterating recursively through the ancestry of a Changebox to find the latest variant of a primitive element can become very time consuming as soon as this ancestry grows big. A cache for the variants was introduced, trading memory space for execution time. After the first lookup for an element in a given scope the found variant is put into the cache. On any subsequent requests in the same scope the cached variant can be used, which results in a major speed up. Performance details are discussed in Section 4.1.2.

3.4.3 Dispatching Message Sends

In order to execute methods depending on a given scope, a mechanism to dispatch message sends to the right variants is needed. In Smalltalk, several techniques for message passing control exist [Duca99]. Methods are stored in the `MethodDictionary` of the class they belong to. This dictionary associates the method selectors with their `CompiledMethod`s. When a message is sent to an object, the value belonging to the method selector is looked up in the method dictionary. If the dictionary does not contain an instance of `CompiledMethod` for this key, the message `#run:with:in:` is sent to the found object. Replacing `CompiledMethod`s by objects to intercept this message is known as *method substitution*.

Using method substitution, it is possible to alter the behavior of a message send and to control the method interface published by an object. In the implementation of Changeboxes, an instance of `MethodElement` representing the actual method takes the place in the method dictionary. It is put there when a change of the corresponding method is captured. `MethodElement` implements the method `#run:with:in:` to perform a lookup of the active method variant for the current execution scope. Additionally, the `MethodDictionary` is modified to only show those selectors of a class that contain a variant in the current scope. Like that, the interface of a class can vary between different execution scopes. In the case that a method was never affected by a change captured by Changeboxes, the `CompiledMethod` remains in its place in the method dictionary and is called directly, without the need for additional dispatching.

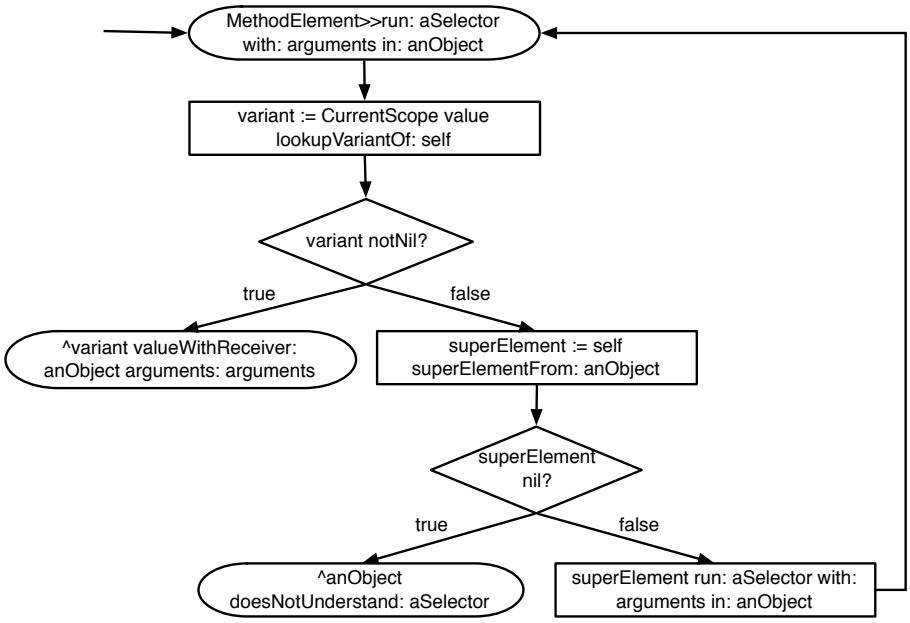


Figure 3.12: Object-oriented message send with a lookup in the current execution scope.

To achieve a behavior identical to object-oriented message sending in a dynamically typed environment like Smalltalk, the algorithm modeled in Figure 3.12 is followed by a `MethodElement` to dispatch a message send.

The main difference to a conventional message send algorithm is the lookup performed for the active variant in the current scope. When a valid `CompiledMethod` is found, it is sent the message `#valueWithReceiver:arguments:` to execute it [Duca99]. The value obtained from this call is returned to the sender of the original message.

If the lookup returns `nil`, the method does not exist for the corresponding class in the current execution scope (either because it was removed in a captured change or because it never existed at all). In this case, the algorithm continues in the `MethodElement` for the same selector in the superclass of the concerned object. This process works up the inheritance hierarchy recursively. When the root class is reached without a non-`nil` method variant being found, the `#doesNotUnderstand:` method is sent. For conventional objects, this causes an exception to be raised.

3.4.4 Dispatching Class References

As opposed to method lookup, class reference resolution in Squeak is not dynamic. All references to classes in the source code are resolved at compile time and a pointer to the actual `Class` object is stored in the corresponding `CompiledMethod`. When a class is rebuilt, the previous class object is replaced using the `#become:` primitive. This causes all variables in the entire system that used to point to the previous class variant to point to the newly created instance. This process is known as *identity swapping*.

While this approach is perfectly sufficient for the Squeak environment, it does not fulfill the requirements of Changeboxes. As with methods, we need to be able to dynamically dispatch the references to a class. Whenever a message is sent to a certain class, it should be sent to the active variant in the current execution scope. The instantiation of an object should depend on the current shape of the corresponding class. To achieve this behavior, two issues needed to be resolved. The first problem is to keep several variants of the same class in the image, the second is the dynamic lookup for the active variant in a given execution scope.

Keeping Several Variants of the Same Class

To retain different versions of a class in a system, the reflective kernel for building classes had to be modified. Identity swapping was removed from the class building process, so that the previous version is not affected. Additionally, not all changes to a class variant require to rebuild it in the conventional Squeak system (*e.g.*, adding a class variable). Because Changeboxes store variants that should not be modified later on (Changeboxes are immutable), a new variant has to be created for every class change. The reflective system facilities were adapted to this functionality.

In Squeak, classes are stored in the global variable `Smalltalk`, which is an instance of `SystemDictionary`. Whenever a class is modified in the scope of a work session, the resulting class variant is encapsulated in the newly created Changebox and the corresponding `ClassElement` is put at the place in the `Smalltalk` dictionary (identical to the `MethodElement` in the `MethodDictionary`). The `ClassElement` takes the role of a dispatcher, as explained in the following section.

Dynamic Class Reference Resolution

With identity swapping deactivated, the class pointers stored in `CompiledMethod`s would always point to the class variant active at their compile time.

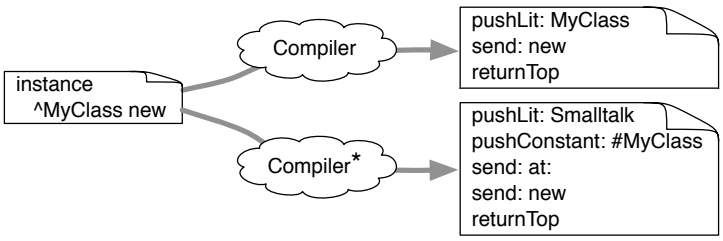


Figure 3.13: The bytecode for a class reference generated by the conventional Smalltalk compiler (top) and by our modified version (bottom).

Thus, a running method would possibly send its messages to old variants, ignoring any more recently built ones. Therefore, class reference resolution must happen at runtime.

In order to achieve the desired behavior, we chose to modify the method compilation process. Instead of putting a direct pointer to the `Class` object in the literals array of a `CompiledMethod` (which is the conventional behavior of Squeak [Guzd01]), we substituted each occurrence of a class in the source code with a lookup of the class name in the global variable `Smalltalk`. The resulting bytecode of this compiler modification is illustrated in Figure 3.13. This modification was possible by using the packages *AST* [AST] and *NewCompiler* [NewC, Denk06], which provide a Smalltalk compiler with high-level syntax tree abstractions.

The `SystemDictionary` was adapted to work with `ClassElement`s, which lookup the active variant of a class for the current execution scope. The design we applied for the `MethodDictionary` could be reused identically. If the lookup for a class variant returns `nil`, a custom block is evaluated, which defaults to raising an exception.

3.5 Merging Changeboxes

A Changebox can build on several ancestors, combining all the changes they contain. This may lead to various conflicts about the visible variants in the merged execution scope. Generally, a conflict occurs when two or more Changeboxes belonging to different ancestries contain different change specifications for the same element. Additionally, removal specifications for certain elements can conflict with specifications for dependent elements. For example, the removal of a class conflicts with the definition of one of its member methods.

The present section discusses the detection and resolution of such merge

conflicts. As Changeboxes capture change processes in the form of change specifications, the concepts of *operation-based merging* [Lipp92] are followed widely. A *pre-processing* of the change specification set to be merged removes any pseudo conflicts (Section 3.5.1). The remaining conflicts are then resolved by an appropriate *merge strategy* (Section 3.5.2). Finally, to obtain a runnable system, the merged specifications have to be sorted and applied based on their dependencies (Section 3.5.3).

3.5.1 Pre-Processing Change Specifications

Operation-based merging builds on the same prerequisites as Changeboxes: not the effects of a change, but their semantics are captured. Changeboxes also model changes with objects instead with pure text files. The encapsulated change specifications correspond to the transformations in operation-based merging. Commutation of specifications can be determined quite easily when the affected elements are considered. Hence, many of the concepts of operation-based merging could be reused for Changeboxes.

For each Changebox to be merged, all change specifications from its ancestry are collected. This entire set is then searched for conflicts before the resolved specifications can be applied. If change specifications for the same element are found in different ancestries, they might conflict with each other.

An additional pre-processing, as suggested by [Lipp92], may be able to remove redundant change specifications. A definition of an element is redundant, for example, when the same element gets deleted later on in the same work session. This pre-processing speeds up the merge process and avoids unnecessary conflicts.

If an ancestry contains several specifications for the same element, only the last one is considered. In Figure 3.14, the specifications for `A>>o` can be consulted to illustrate this policy. This method was first defined in the lower branch and then removed again. The second specification overrides the first one, which can therefore be discarded from the specification set. The pre-processing removes unnecessary change specifications in this case.

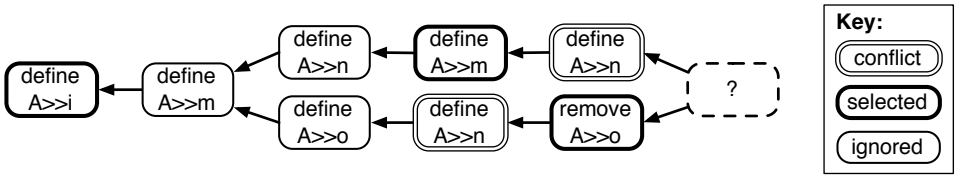


Figure 3.14: Merge of two Changeboxes with common ancestors.

Different change specifications for the same element in different ancestries do not necessarily have to present a conflict. Having the directed acyclic graph structure of Changeboxes available helps one to detect some pseudo conflicts that would emerge from comparing only two versions without looking at their common ancestor. Consider the situation of `A>>m`. A lookup in both branches results in two different definition specifications which conflict with each other at first glance. Taking the graph structure into account, we see that the first definition of `A>>m` is a common ancestor of both branches. No changes to this method were performed in the lower branch. It was only overridden in the upper one. Therefore, the common change specification can be discarded and only the specification from the upper branch remains. This resolution of a pseudo conflict corresponds to a simple three-way merge.

Two different specifications exist for `A>>n`. Because they were defined separately in both branches, neither of them can be discarded. They remain in the set of change specifications and present a conflict to be resolved in a later step.

As three-way merging is a practice generally agreed upon [Mens02], change specifications are always pre-processed with this technique. However, three-way merging only eliminates pseudo conflicts. Other conflicts may still remain, and no common resolution approach would meet all the needs for different domains.

3.5.2 Merge Strategies

Potential conflicts in the pre-processed set of change specifications have to be resolved before the specifications can be applied to generate runnable elements. For certain applications, the resolution policies offered by operation-based merging could be too sophisticated and purely autonomous decisions would be required. Or a simple order based procedure would be sufficient in another domain. To comply with these different needs, the Strategy pattern [Gamm95] was used for the merge process. Every merge is controlled by a customizable `MergeStrategy` that provides a tailored conflict resolution policy and appropriate semantics. This keeps the Changebox model both simple and extensible.

A `MergeStrategy` gets the set of pre-processed change specifications and their originating Changeboxes as input and is expected to return a non-conflicting subset of these specifications (definable by `#setConflicting:` and `#resolveOpenConflictsFor:` as shown in Figure 3.15). When an element possesses several simple change specifications, *i.e.*, for definition or removal, they conflict amongst each other. Such conflicts should be resolved

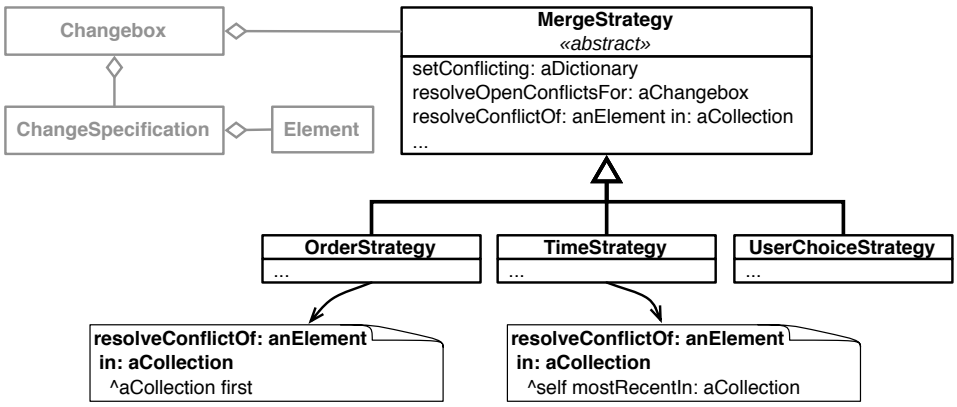


Figure 3.15: The abstract `MergeStrategy` and its concrete subclasses.

by the `#resolveConflictOf:in:` method. Conflicts that occur amongst specifications for different elements have to be identified and resolved by the `MergeStrategy` independently. As a proof of concept, the following merge strategies were implemented:

Sort Order Strategy. The Changeboxes to be merged are specified in an ordered collection. In the case of a conflict, this strategy chooses the change specification found in the first Changebox (or in any of its antecedent Changeboxes) and discards the others. Conflicts occurring amongst different elements are ignored.

Latest in Time Strategy. When several change specifications for one element are found, this strategy chooses the one from the Changebox that was created last in time. This strategy illustrates the use of meta-information for the conflict resolution process. Any inter-element conflicts are ignored.

User Choice Strategy. This strategy acts as an example for a more sophisticated conflict resolution. It follows the principles of revision control systems and provides the most appropriate resolution for software systems developed by multiple programmers. The user is presented a list of all conflicting change specifications and is supposed to choose one of them. This task is supported by a graphical user interface as shown in Figure 3.16.

Conflicts occurring between different elements are presented to the user for resolution as well. Only when all conflicts are resolved will the merge process continue. The user has the possibility to cancel the merge operation at any point.

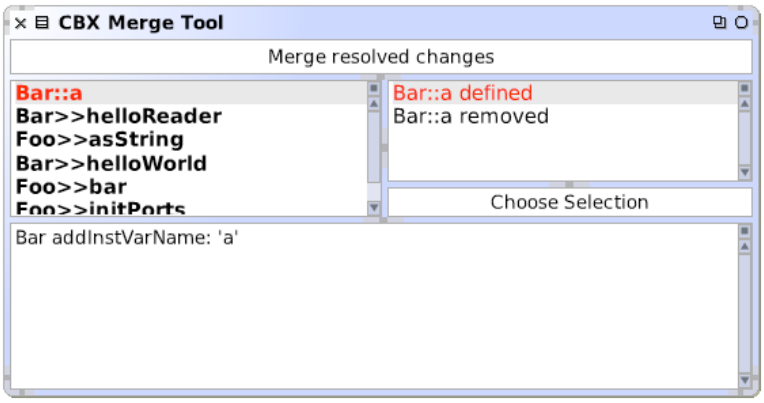


Figure 3.16: User interface of the user choice merge strategy.

3.5.3 Change Specification Dependencies

In order to have the merged Changebox be runnable, the resolved change specifications have to be applied to generate the appropriate classes and methods. Because the specifications merged from different branches can define elements in a shape not existing in any of the branches, the previous variants cannot simply be reused. For example, adding different instance variables to the same class in each branch can result in a merged class with all these instance variables, requiring a re-compilation of all its methods as well.

After all conflicts are eliminated, the remaining change specifications are sorted based on their dependencies, just as in operation-based merging. This order is determined by the actual specifications and their changed elements. The following order of application is proposed:

1. Definitions and removals of classes. Superclasses must be defined before subclasses. Class removals invalidate potential specifications of adherent fields and methods.
2. Definitions and removals of fields.
3. Definitions and removals of methods.
4. Refactorings for classes.
5. Refactorings for fields.
6. Refactorings for methods.

This order is based on the following dependencies. When defining a field or a method, the corresponding class has to be present in the system. Hence

class specifications must be applied before any other specifications. The class hierarchy is important, since class definitions requires the existence of the according superclass. When a class is removed, none of the adherent elements can be modified anymore. Although this should be treated as a conflict and be resolved by the merge strategy, invalidating the corresponding specifications guarantees a fail-safe behavior.

Steps 1 and 2 could be combined in order to increase performance. Instead of building a simple class and iteratively adding field variables to it, a class with all the specified fields can be created in one single step.

Because method compilation takes the offsets of instance variables into account, the associated class with all its fields should be created before. For handling class references in methods, the according class must already exist as well. Hence, methods specifications are applied third.

Refactorings work generically on present elements and are applied only after the simple specifications. A renaming specification for a class, for example, results in a class that belongs to a different `ClassElement`. Method specifications for the old class name would miss their associated class when applied after the class rename. Therefore, we assert that all elements are present before performing any refactorings. Following the considerations above, refactoring specifications for classes are applied first, then the ones for fields and finally those for methods.

The merge process terminates when all selected change specifications are applied. A Changebox representing the merge holds all generated class and method variants, providing a new scope for execution. The semantics of the operation are covered by the chosen merge strategy.

3.6 Tool Support

Various tools have been implemented or adapted for the use with Changeboxes. This section presents an overview of the most common ones used for the daily work.

3.6.1 Work Session Browser

To manage the various work sessions, a new browser had to be built. The Work Session Browser allows the user to inspect the existing work sessions and their assigned Changeboxes, invoke designated functions on them and execute code in the respective scopes.

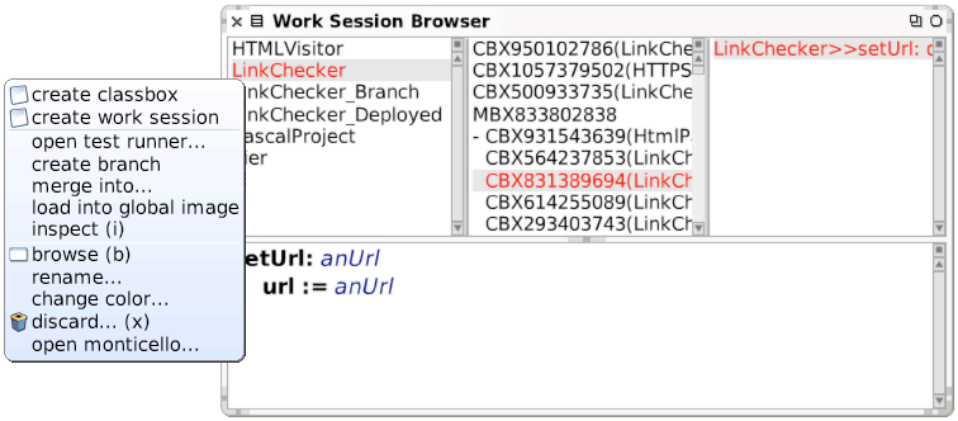


Figure 3.17: Screenshot of the Work Session Browser.

In the left-most pane displayed in Figure 3.17, a list of all currently active work sessions can be found. Each work session is given a name and a color that are used to draw work session aware windows, which makes it easier to separate them. Over a context menu, new work sessions can be created, existing ones can be branched, merged with others, loaded into global scope, modified or removed. Additionally, the tools described in the following sections can be opened from here, whereby the selected work session is passed as the tool's execution scope.

The central pane contains a tree of Changeboxes, originating from the Changebox currently defining the scope of the work session. The Changeboxes are sorted chronologically, with the most recent one on top, to represent the history of the work session. After every Changebox that represents a merge, the following Changeboxes are indented and every direct ancestor of the merging Changebox is marked to emphasize the start of another ancestry. This representation corresponds to commonly-known, fully expanded tree views. Something that is not yet modeled here is the branching of work sessions. Whenever a work session is split up into branches that are merged together again later on, the initial Changeboxes appear twice in this simulated tree view, once for each branch. So this representation is an attempt to fit the directed acyclic graph structure modeled by Changeboxes into a list view. Far from being perfect, it allows the user to browse through the Changebox history in a well-known environment. Similar actions as for the work session can be invoked on the elements in this list. The text pane below the three columns is used to display additional information about the selected Changebox, such as the author and the time of the change.

The last pane in the upper row displays the change specifications held by the currently selected Changebox. The modified element and a short description

of the change type are used to identify the specifications. A complete representation of the change specification is presented in the lower pane. This representation usually contains the source code for definition specifications, *i.e.*, the method source or a class creation statement, and an appropriate description for removal and renaming specifications.

Additionally, any code that is executed in the pane in the lower half of the window runs in the scope of the selected work session or Changebox. This provides a convenient way to go back in time and run selected statements in different execution scopes and watch the changed behavior. Thus, the Work Session Browser acts as a first place to get familiar with the scoped execution of code and is the entry point to the other tools.

3.6.2 Developing: OmniBrowser

OmniBrowser [Putn] is a rewrite of the conventional system browsers in Squeak. It provides about the same functionality and is designed for flexibility and extensibility. It proved to be convenient for integrating Changeboxes, so it was preferred to act as the main code browser. We made OmniBrowser aware of work sessions, allowing a programmer to browse and develop a system in several trails.

From the Work Session Browser, OmniBrowser windows can be opened for a given work session. The view provided by the modified OmniBrowser

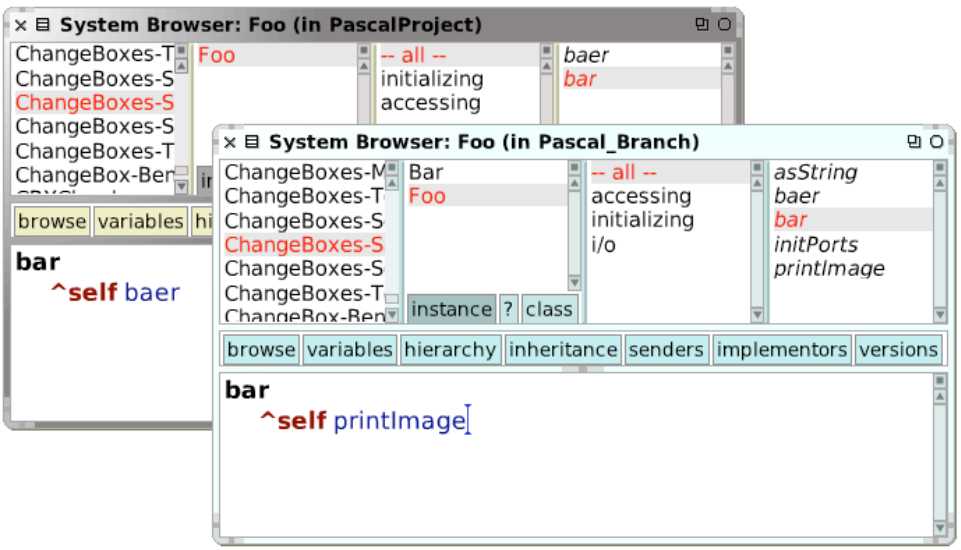


Figure 3.18: OmniBrowser windows for different work sessions.

reflects the system in the scope of this work session. Figure 3.18 displays two windows with different work session. All actions invoked from a browser are performed in the execution scope of the corresponding work session. New browser windows sharing the same work session may be opened. All changes are captured by the work session as explained in Section 3.3.2.

All additional browsers work as expected. Implementors, senders, hierarchies, inheritance, references and variables can be browsed in the scope of a given work session. The browsers are all colored and named after the work session they hold, so the user will not lose the overview over the different windows usually opened in Squeak. For the programmer, there are no differences between developing with Changeboxes or with a conventional system.

3.6.3 Source Control: Monticello

Monticello [Brya] is a versioning system for Squeak that provides support for concurrent development and is able to work with different source repositories. We extended Monticello to be Changebox-aware. Snapshots of packages, the bundling unit of Monticello, can be loaded into work sessions and vice versa. This allows a programmer to use external source repositories for code developed with Changeboxes as well. For example, external packages could be loaded in a sandbox work session, without affecting the remaining image.

An additional feature we implemented is the subsequent loading of multiple versions into one work session. Either a selected subset of all available snapshots for a package is loaded, or all snapshots in their sequential order. This feature may be used to study the evolution of a package post facto, with all its changes encapsulated in Changeboxes (see Chapter 4).

3.6.4 Testing: Test Runner & Debugger

Unit testing is an important activity during software development [Beck98]. With SUnit, Smalltalk provides a basic testing framework that was adapted for many other programming languages. The graphical user interface of SUnit, the Test Runner, was adapted to be aware of execution scopes as well. Tests cannot only be run in the work session a programmer is currently developing, but in the scope of any previously recorded Changebox. Evolutionary analysis of test cases and test results becomes possible.

Another important tool in relation to the Test Runner is the Debugger. In Squeak, a debugger pops up whenever an unhandled exception is encountered and provides operations to step through code and to modify it in

place. As a debugger is running in a different process than the interrupted one, it does not share the same execution scope. Stepping, which simulates the interrupted process in a different thread, had to be modified to use the appropriate scope.

Modifying code inside the Debugger could lead to ambiguities as soon as a special execution scope is defined for the interrupted process. Should the change be appended to the work session, or does it affect the global scope? This issue has to be solved by the user, who is presented an option dialog whenever a method that is interrupted in a non-global scope is saved.

Chapter 4

Evaluation

To validate our model and implementation, we performed several experiments in different application domains. First of all, we were interested in the performance of our implementation and conducted various benchmarks. The results for different real world applications are presented in Section 4.1. The fact that Changeboxes have all versions of a system directly runnable was exploited in Section 4.3 to analyze the evolution of a protocol library. The runtimes of different snapshots could be easily compared in order to identify the changes responsible for performance decreases. A completely different application of Changeboxes was examined by using them to model classboxes. How this module system supporting local rebinding could be emulated is discussed in Section 4.2. These various uses of Changeboxes outline only a part of their possibilities offered, but clearly show the value of this unifying approach for modeling software change.

4.1 Benchmarks

The current implementation of Changeboxes builds on the kernel libraries in the image and does not contain any changes to the Squeak virtual machine. It did not undergo any significant optimization processes and leaves open several possibilities for such efforts. To obtain a general idea about the execution speed of systems using Changeboxes and to identify the areas where performance could be improved, we conducted several benchmarking experiments.

Various aspects and applications of Changeboxes were tested. In Section 4.1.1, the use of Changeboxes for real projects is examined under the benchmark of execution time. Section 4.1.2 reveals the pure values for basic operations like messages send and class reference resolution. It can be shown that real

world projects execute with an overhead of 4% to 500% although the basic operations suffer from a bigger speed decrease.

4.1.1 Real World Applications

To get an impression of the performance of a system using Changeboxes, two real world applications have been loaded into Changeboxes. This was achieved by subsequently loading all available versions from a source code repository. Although this does not represent the real development trail of an application, a large line of subsequent Changeboxes is created, encapsulating the elements of all versions from the source control repository.

The focus of our experiments is on the performance of a system using Changeboxes compared to the same system not using Changeboxes. As an indicator we used the runtime of the unit tests of the applications, as they should cover most of the application code.

All benchmarks were performed on a MacBook Pro 2 GHz Intel Core Duo with 2 GB RAM. The Squeak virtual machine version was 3.8.12beta4U.

Hessian

The first application we benchmarked is Hessian, an implementation of a binary web service protocol. Source control contained 13 versions for this project, which resulted in a total of 570 Changeboxes. The last version of Hessian contained 28 classes, 468 methods and 107 test cases. The running time for these tests in a plain Squeak image is 3.85 seconds. The results of this experiment are shown in Table 4.1.

# of Changeboxes	first execution		subsequent execution	
	time	ratio	time	ratio
570	4.33 s	1.12	3.96 s	1.03
14 820	44.51 s	11.55	4.05 s	1.05
29 070	67.73 s	17.57	4.09 s	1.06
43 320	85.35 s	22.15	4.03 s	1.05
57 570	112.13 s	29.09	3.99 s	1.03
71 820	145.34 s	37.71	3.99 s	1.03
86 070	173.12 s	44.92	4.02 s	1.04
100 320	204.71 s	53.12	4.03 s	1.05

Table 4.1: Hessian: runtime in seconds of 107 tests with 570 Changeboxes and artificially added Changeboxes. Runtime without Changeboxes: 3.85 s.

To highlight the impact of the number of Changeboxes on the element lookup, we artificially added intermediate Changeboxes with a dummy change between the real Changeboxes. Because the ancestry is larger with artificially added Changeboxes, but no runnable elements are found in the dummy Changeboxes, the lookup has to go deeper and thus takes a longer time to complete. The first row shows the values for the original 570 Changeboxes. In the subsequent ones 25, 50, 75, 100, 125, 150 and 175 dummy Changeboxes have been inserted in between each connected pair.

For the initial run, which is shown in the first column, the variants have not yet been cached and thus a lookup over the complete ancestry has to be performed for every element request (*i.e.*, message sends and class references). The run with the original 570 Changeboxes takes 4.33 seconds, representing an overhead of 12% (*i.e.*, a ratio of 1.12 compared to the time in a conventional image). For the second and all subsequent executions, the variant cache is filled and no class and method lookup is required anymore. The overhead of the test run drops to 3% (*i.e.*, a ratio of 1.03).

In the following rows, the time grows linearly with the number of Changeboxes in the ancestry and becomes 53 times slower with over 100 000 Changeboxes in place, compared to the running time of the tests without Changeboxes. This overhead is very high, but it only applies for the initial access of every element. For all subsequent runs a constant time could be achieved, independent of the number of Changeboxes. The differences occurring in the corresponding column conform with regular fluctuations. The subsequent execution time represents an overhead of about only 4%.

Given the possibility to fill the cache in the background before execution, the resulting overhead when using Changeboxes proves to be really small. For this application, the use of Changeboxes is absolutely acceptable.

Pier

To obtain data for a medium project, we performed a similar experiment with Pier, an object-oriented content management system [Reng06]. The repository of Pier contained 115 versions, the last one consisting of 194 classes and 1883 methods. In total, 6283 Changeboxes were created from these versions. The 1057 tests of Pier run in 1.01 seconds (average time over 100 runs) in a plain Squeak image.

The values in Table 4.2 show similar results as for the Hessian implementation - a linear growth of the runtime for the first execution and a constant time once the caches are filled. However, the overhead for running the Pier tests with a filled cache is much bigger with a factor of about 4.9 compared to a system running without Changeboxes. The ratios for the first execution

# of Changeboxes	first execution		subsequent execution	
	time	ratio	time	ratio
6 283	24.28 s	23.96	4.87 s	4.83
56 547	305.40 s	301.37	5.00 s	4.94
106 811	489.63 s	483.18	5.02 s	4.95

Table 4.2: Pier: runtime in seconds of 1057 tests with 6283 Changeboxes and artificially added Changeboxes. Runtime without Changeboxes: 1.01 s.

with roughly 50 000 and 100 000 Changeboxes are about 10 times higher compared to Hessian.

Comparing Hessian and Pier

These runtime overhead differences between various projects tell more about the applications themselves than about Changeboxes. Only methods and classes belonging to the respective application are encapsulated in Changeboxes and have to be dispatched. The objects belonging to the systems libraries are unaffected and still run at the same speed.

Hessian spends a lot of its execution time with encoding and decoding binary data. Most of these conversions build directly on the functions provided by the appropriate data types from the system framework (*e.g.*, `String` and `Array`). Only a small layer of application code encapsulated in Changeboxes is traversed and needs to be dispatched, the main work is then done by low-level system functions. Therefore, the overhead for the thin application layer encapsulated with Changeboxes is marginal.

Pier, on the other hand, models many different entities and provides the behavior to work with them. Most of the message sends are within the objects of the application and thus involve a variant lookup. As a result, the overhead is much higher.

The runtime overhead of Changeboxes depends strongly on the actual application. With a project like Pier which builds on many local message sends, an implementation using Changeboxes is about five times slower than in a conventional system. Since the current implementation is at the state of a proof of concept, the obtained values are more than adequate.

4.1.2 Micro Benchmarks

Besides examining the cost to run real projects, execution times for the most basic operations were measured. In Smalltalk, these operations are message

Operation	Global scope	Dynamic Variables		`Process` Instance Variable	
	Time	Time	Ratio	Time	Ratio
10^6 message sends	100 ms	14 023 ms	140.2	6 510 ms	65.1
10^6 class lookups	2 030 ms	13 724 ms	6.8	6 053 ms	3.0

Table 4.3: Benchmarks for basic operations, average from three runs.

sending and class reference resolution. In a conventional image, message sends are directly performed by the virtual machine, and class reference resolution is accomplished at compile time, resulting in no runtime costs (apart from pushing the class on the stack). As discussed in Section 3.4, both of these operations are dispatched at runtime in our implementation. The basic operations were benchmarked using different implementations of the scoping mechanism.

The micro benchmarks were conducted with few Changeboxes, containing only the definitions of the reference class and method used for the experiments. The variant cache was filled before starting the measurements. Like this, the pure overhead of the basic operations could be measured independently from the scope lookup. As we could show in the previous section, the execution time is independent of the Changebox lookup ancestry depth when using a cache.

Table 4.3 shows the results of the benchmarks for the basic operations. We compared times for message sending and class lookup (i) in the global scope (*i.e.*, without Changeboxes), (ii) using dynamic variables for retrieving the current scope object and (iii) with the scope object stored in an instance variable of the active `Process` object.

The large difference between message sends and class lookups in the global scope originates in the compiler modification introduced for Changeboxes. While message sends in the global scope operate at the same speed as in a conventional image, class reference resolution always happens at runtime instead of at compile-time. In a plain system, class access works with no cost because a direct reference is present in the method's literal array. The compiler modification affects all class references in a system, independent of the presence of a special scope. The Squeak virtual machine is neither designed nor optimized for runtime class resolution, hence the slow time.

The big overhead of message sends when using Changeboxes is not surprising as it reflects the difference between executing compiled machine operations and interpreting them in a high-level language.

This experiment also revealed a factor two difference between dynamic variables and the process instance variable for basic operations. This issue was

Scope Object Access (10^6 times each)	Dynamic Variables	`Process` Instance Variable
Stack depth 0	7 455 ms	258 ms
Stack depth 100	7 967 ms	258 ms
Stack depth 1000	17 833 ms	255 ms

Table 4.4: Benchmarks for scope access, average from three runs.

investigated further. Table 4.4 shows the access times for the scope object based on different method call stack depths. This is important because dynamic variables have to search that stack upwards for the definition of their value. The method execution context stack was built up using a recursive function.

The difference between the two approaches is even more significant when measured in isolation. Even with a flat execution stack, accessing the `Process` instance variable is 29 times faster than accessing a dynamic variable. With growing method context stacks, dynamic variables get increasingly slower, while the access time for the `Process` instance variable remains stable. The values obtained from this experiment lead to the decision to use a `Process` instance variable to define the execution scope.

10^6 message sends take 6510 ms to run using the process instance variable (s. Table 4.3). To identify the areas where future performance increases would be possible, we split up our modified method lookup (as described in Section 3.4.3) into the following measurable steps:

1. Dispatch the message send on the object in the method dictionary.
2. Determine the scope object from the `Process` instance variable.
3. Lookup the active variant of the method in the current scope.
4. Call the found method variant.

The first step only takes a fraction of the total time with 146 ms (2.2%). The scope object lookup takes 258 ms (4.0%) as shown in Table 4.4. The third step for the variant lookup in the cache uses the most time with 6006 ms (92.3%). Calling the found method variant then takes 100 ms (1.5%).

The first step is fast because it uses a reflective capability of the virtual machine. The second step only accesses instance variables and is also not time consuming. The last step corresponds to a single message send and is also negligible. The variant lookup has the main impact. It consists of many message sends including a dictionary lookup in a cache implemented in Smalltalk. Possible ways to further improve this performance are discussed in Section 5.4.1.

4.2 Modeling Classboxes

Changeboxes can model spatial and temporal dimensions of software change. With multiple simultaneously active Changeboxes denoting execution scopes in a running system, different views of the same software artifacts are possible. An artifact may exhibit a different behavior or shape depending on the current execution scope. In this section, we illustrate this possibility by implementing Classboxes with Changeboxes.

Bergel proposed Classboxes [Berg05a], a module system for object-oriented languages supporting local rebinding. Local rebinding means that changes made by a classbox are only visible within that classbox or classboxes importing it. We show how Classboxes may be expressed as a semantic addition on top of Changeboxes. As a proof of concept, we implemented a Classbox example first introduced in [Berg03a].

4.2.1 The Classbox Model

A classbox is defined as follows:

> "A classbox is a module containing scoped definitions and import statements. Classboxes define classes, methods and variables. Imported declarations may be extended, possibly redefining imported methods." ([Berg05a], p. 39)

All definitions made in a classbox and its imported classes are globally accessible by all methods executed in the scope of that classbox. Imported classes may be extended within a classbox. The extensions are then only visible within that classbox, but not in the classbox originally defining the class. As a consequence, different versions of the same class may be active within a single running system. Any visible class — imported or defined — in the scope of a classbox may be imported from another classbox.

To illustrate the mechanism of classboxes, we use the implementation of a HTML link checker that is built with several classboxes, as illustrated in Figure 4.1. The classbox *HTML CB* defines a module that parses a HTML document into an abstract syntax tree. The elements of this tree are extended in the classbox *HTMLVisitor CB* (that imports *HTML CB*) to accept visitors. An abstract `HTMLVisitor` class is created in this classbox as well. When code is running inside the scope of classbox *HTML CB*, the methods `#acceptVisitor:` of `HTMLEntity` and its subclasses are not visible. Classbox *LinkChecker CB* imports *HTMLVisitor CB* and defines a subclass of `HTMLVisitor`, `LinkChecker`, which overrides the method `#acceptAnchor:` to check for dead links. For this purpose, we make use of the method `#httpGetDocument:` from class `HTTPSocket` from the global

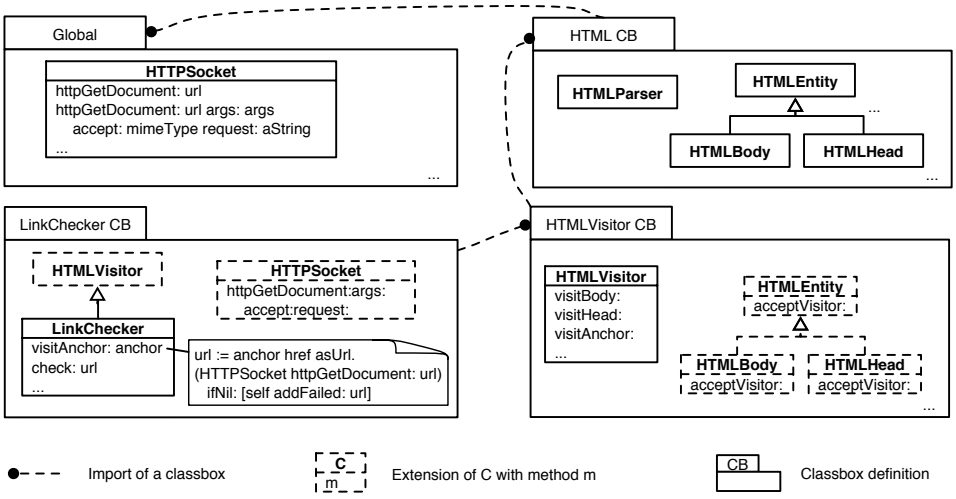


Figure 4.1: A link checker modularized with classboxes.

classbox. This method calls `#httpGetDocument:args:accept:request:`, which returns the HTML document or the HTTP response from the server in case of an error as a stream. For our link checker example, the effort to parse the stream in order to detect an error message from a server seemed inappropriate. Instead, that method was modified to return the HTML document or `nil` if any error occurred, which was easier because the HTTP error codes are directly available there. This modification again is only visible within the classbox *LinkChecker CB*, any other clients of `HTTPSocket` remain unaffected.

Classboxes offer a model to control the scope of class extensions. The properties provided by this model are summarized as follows:

Class Extensions with Redefinition. Classes imported from other classboxes may be extended by redefining and adding new methods within the local classbox. This is illustrated by classbox *HTMLVisitor CB* that adds the method `#acceptVisitor:` to `HTMLEntity` and all of its subclasses. The implementation of classboxes in Smalltalk does not provide removal of methods or other changes on classes and fields.

Locality of Changes. The visibility of redefinitions is confined to the classbox that made the changes. Classes in other classboxes are not affected. A classbox that imports *HTML CB*, but not *HTMLVisitor CB* would obtain a `HTMLEntity` tree from `HTMLParser` that is not visitable.

Local Rebinding. A system is executed in the scope of a certain classbox. Message sends always run the methods defined in this classbox, even if they originate from a different classbox. For example, the method `#httpGet-Document:` is called from within the *LinkChecker CB* classbox. This method is not part of *LinkChecker CB*, but it calls the method `#httpGetDocument:-args:accept:request:`, which was redefined there. When running in the scope of *LinkChecker CB*, the redefined method is executed. When another classbox defines the execution scope, the original method is run. Hence, classboxes are *reentrant*. *Local rebinding* asserts that the correct version of a method is executed. From within a classbox, the world looks flat, *i.e.*, there is only one visible version for every method.

4.2.2 Using Changeboxes to Express Classboxes

A Changebox may be directly mapped to a classbox. Although a Changebox only encapsulates a single change, including its ancestors, a complete executable scope of software elements is defined. This scope is directly comparable to the scope of a classbox. Whenever a change is performed, a new snapshot of the classbox represented by a new Changebox is obtained.

The properties described in the previous section are all met by Changeboxes. Classes defined in a system snapshot represented by one Changebox may be extended by (re-)defining selected methods in a subsequent Changebox. This illustrates *class extension with redefinition* and even provides the additional possibilities to modify fields and remove methods. The *locality of changes* is a crucial concept for Changeboxes. Changes are encapsulated in Changeboxes and do not affect other parts of the system (*i.e.*, different Changeboxes). When executing code in the scope of a Changebox, the versions defined in this scope are used, independent of the versions visible at the point the currently executing method was defined. Changeboxes thus provide *local rebinding*.

With the concept elaborated until here, the scope of a classbox is represented by Changeboxes, but imports are still missing in this model. We found that a classbox importing another one corresponds to a merge of two (or more) Changeboxes, using an appropriate merge strategy. The semantics behind a classbox import say that all elements defined in the local classbox should override the same elements from an imported classbox. This is needed to fulfill the property of *class extension with redefinition*. A merge strategy (named `ImportStrategy`) reflecting these semantics was implemented. This proved to be straightforward. For each element conflict the specification originating from the local Changebox is chosen and the one from an imported Changebox is discarded. If several classboxes should be imported, an order has to be specified to define their respective priorities.

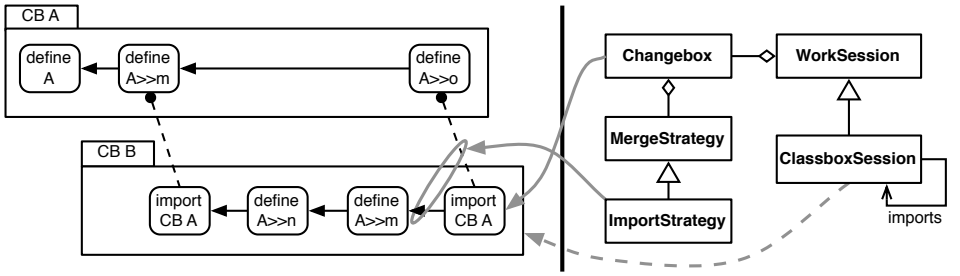


Figure 4.2: Two classboxes modeled with Changeboxes.

The left part of Figure 4.2 illustrates how we modeled classboxes using Changeboxes. *CB A* initially defines a class `A` with a single method `#m`. This first version is imported into *CB B*, where a Changebox is created to represent this import (*i.e.*, a merge using the `ImportStrategy`). In the following, *CB B* defines an additional method `#n` and overwrites `#m`. A call to `A new m` from *CB B* will involve a lookup for `#m` that resolves to the local definition. Hence, `A` is extended by `#m` and `#n` in *CB B*. Both methods are not visible from the latest Changebox representing *CB A* (*i.e.*, the one that initially defined `#m`).

To reflect changes from an imported classbox, they need to be merged into the importing classbox continuously. This can easily be achieved by a tool that keeps track of changes in classboxes and performs the merge automatically. The most recent version of a classbox might be tracked by a work session, hence a subclass of `WorkSession` named `ClassboxSession` was created. `ClassboxSession` is not required to model classboxes, it only acts as a *helper tool* for automating certain tasks. The design of this extension is shown on the right hand side of Figure 4.2. Changes performed in a `ClassboxSession` are considered to be performed in the managed classbox. The Changebox representing the last version of the classbox is always retained and will be used as an ancestor for subsequent changes. The imported classboxes (managed by their own `ClassboxSessions`) are cached in the local `ClassboxSession`, as well as the last merged (*i.e.*, imported) Changebox from each imported classbox. Whenever code should be executed in the scope of a `ClassboxSession`, the last merged Changeboxes are compared to the current Changeboxes of previously imported `ClassboxSessions` and are merged automatically with the local one if they do not match.

When the Changebox last displayed in *CB A* (defining `A>>o`) is created, the `ClassboxSession` managing *CB B* will merge it with its most recent Changebox (*i.e.*, the one overwriting `#m`). With the special merge strategy, only elements not present in *CB B* will be merged from *CB A*. This affects

method `#m`, which is defined in both classboxes now. The merge strategy gets notified about the two conflicting definitions and chooses the one originating in *CB B*, because local changes take precedence over imports. Method `#o` is not yet existing in *CB B* and it is merged in. Hence the new execution scope of *CB B*, as defined by the rightmost Changebox, will know the newly added method `#o` by *CB A*, without altering any previously defined methods in *CB B*.

The Link Checker Modularized with Changeboxes

Using this lightweight implementation, we were able to rebuild the modularized link checker application with Changeboxes. The HTML parser package was loaded into a first `ClassboxSession` to build the *HTML CB* classbox, using the Changebox-aware Monticello. The second `ClassboxSession` was initially merged with the last Changebox of the loaded *HTML CB* session and then extended with the visitor class and methods. Finally, in a third `ClassboxSession` representing the *LinkChecker CB*, after being merged with *HTMLVisitor CB* (which already contained the Changeboxes of *HTML CB*), the `LinkChecker` was implemented and the required modifications of `HTTPSocket` were performed.

The behavior of this Changebox-based link checker application is identical to the one using the original classbox implementation. The `#acceptVisitor:` methods added in *HTMLVisitor CB* are only visible there and from *LinkChecker CB*, but not from *HTML CB*. This illustrates the *locality of changes*. In the *LinkChecker CB*, `HTTPSocket` was *extended through redefinition* of the method `#httpGetDocument:args:accept:request:`. No other methods nor the class itself had to be touched for this modification. When running in the scope of *LinkChecker CB*, the redefined version of this method is executed. From the global scope, the original version will be called, according to the *local rebinding* property.

The `ImportStrategy` asserts that the correct versions are visible in a classbox. When an element was modified in the local `ClassboxSession`, the according specification will be chosen over others from imported Changeboxes. If an element was not locally defined, the one from an imported Changebox will be merged in. Because imported classboxes have already been merged with their imports in the same way at the point of a merge, the transitive relation of the import operation is abode. Only when no definition specification for a called method is found in a `ClassboxSession` the inheritance relationship of the called object is used. The classbox property that *import takes precedence over inheritance* is met with this approach.

4.2.3 Discussion

In the previous section, we presented an approach to model classboxes using Changeboxes. This solely required the implementation of a special merge strategy which represents the semantics of a classbox import. Because the `ImportStrategy` is able to separate the elements from the local and the imported Changebox (*i.e.*, classbox), the entire import information is contained in the merging Changebox. This information remains available over the ancestry of every subsequent Changebox representing a new snapshot of the same classbox.

This approach meets all major properties of classboxes, including *locality of changes*, *local rebinding* and *class extensions with redefinition*. It is not only possible to extend imported classes with new methods and redefine existing methods, but also to locally remove methods from imported classes, to add and remove instance variables and to perform class modifications only visible in the local classbox. This introduces several new possibilities not available in the conventional implementation of classboxes. Furthermore, Changeboxes not only represent the module composed by a classbox, but also its evolutionary history.

With the extension of the `WorkSession` to a `ClassboxSession`, the merge process reflecting an import of a classbox could be automated. Furthermore, all tools that are aware of work sessions could be directly used with `ClassboxSession`. The present implementation merges lazily, *i.e.*, only when needed for executing the system. Merging constantly on every change would be possible by using an Observer pattern [Gamm95]. The `ClassboxSession` is not required for running or analyzing a classbox represented by a Changebox. All information is already captured by Changeboxes, including the imports. The `ClassboxSession` only serves as a tool to keep track of changes and to update the managed classbox appropriately.

Our classbox model involves minor variations from the conventional classbox model. These variations mostly originate from the design decisions inherent to Changeboxes and are discussed in the following.

Implicit Import

By using the merge functionality of Changeboxes to represent classbox imports, all elements from the scope of a classbox are *implicitly imported*. This allows one to use all available classes directly. The original classbox model requires that for every class referenced the *import is explicitly stated*.

Explicit imports help to preserve inter-class associations and inheritance hierarchies of imported classes. Consider an example in which `B` inherits

from `A` and both classes are defined in classbox *K*. If classbox *L* imports only `B` and defines a new `A'`, `B` still inherits from `A`. Because of the implicit import, `A` would be imported as well with Changeboxes. Redefining it would cause `B` to inherit from the new version `A'`. By modeling classboxes with Changeboxes, associated classes cannot be re-defined separately.

The explicit import property also helps to avoid *diamond conflicts*. They happen when two classboxes extend a class `A` with different methods `#m` and define a uniquely named subclass (`B` and `C`) of `A` each. With explicit imports, only `B` and `C` may be imported in another classbox, preserving the two different extensions of `#m`. A detailed description of this problem is found in [Berg05a]. With implicit imports, a classbox importing the two others imports `A` as well, whereby the two versions of `#m` conflict.

Changeboxes require the import of any element that should be accessible. An import of only `B` and `C` without their superclass `A` would not be possible with Changeboxes. When `A` is missing, the definition specifications for `B` and `C` could not be applied to generate the runnable entities for the merged execution scope. Therefore, every element is imported implicitly with Changeboxes. `A` is imported together with `#m` from both classboxes, which results in a conflict. With the `ImportStrategy`, this conflict is resolved regarding the order of the imported classboxes.

Avoiding diamond conflicts in this way contrasts with the flat world picture brought by local rebinding. In the conventional classbox model, `A>>m` looks differently depending on which subclass of `A` calls this method. This increases complexity and might often be unclear to developers. In a real flat world, methods would always look the same.

Lookup of Classes

It is an inherent part of Changeboxes that not only methods, but also classes depend on the current execution scope. Our model of classboxes inherits this property and therefore provides a *local rebinding of classes*. The lookup of classes is considered only as an alternative approach for classboxes ([Berg05a], p. 54).

This allows one to extend classes not only with methods, but also with fields, such as, for example, instance or class variables. Because class reference resolution is based on the name of a class, it is not recommended to define different classes (*e.g.*, for different domains) in separate classboxes with the same name when using Changeboxes. Equal class names should only be used in classboxes that do not contain any intersecting import chains.

Removing Imports

As a consequence of merging, which is a purely additive function, classboxes once imported cannot simply be removed again, since the ancestry of a Changebox is immutable. For that purpose, tools would have to provide support. For example, a tool could compute the changes required to neutralize the imported specifications, like removing elements that have been defined during a merge process. Another possibility would be to collect all change specifications in an ancestry, following only the local classbox ancestors, and to re-apply them in a new `ClassboxSession`, without the merges being performed.

4.3 Evolution Analysis with Changeboxes

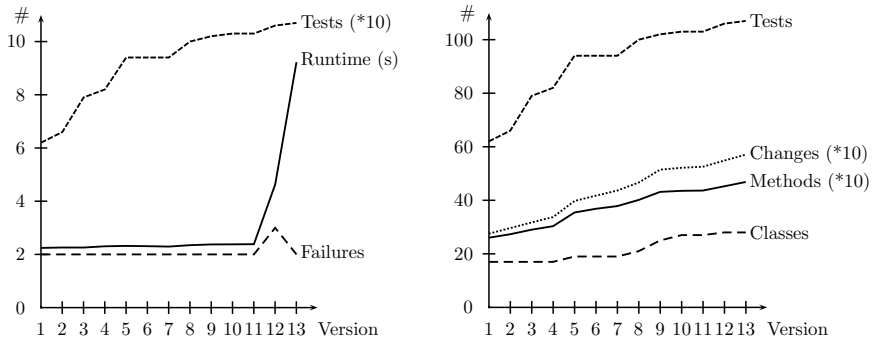
Changeboxes model the entire evolution of a software system at a fine-grained level while every single snapshot is directly runnable. To illustrate the advantages emerging from this model and especially from the directly runnable snapshots, we conducted an evolution analysis of the Hessian protocol. This case study focuses on the temporal dimension of change represented by Changeboxes.

Software evolution analysis often builds on the history of the structural software elements of a system [Girb04b, Girb05a]. Metrics like the number of classes or lines of code for methods are consulted for this purpose. While these values and their evolution are easily determinable with Changeboxes, having all versions of a system concurrently runnable additionally opens up a large field of new analysis methods.

For each of the 13 versions found in the source repository of Hessian [Hess], the unit tests were executed and the number of tests, failures and errors recorded. The results with the corresponding runtimes are displayed in Figure 4.3(a).

We observed an almost constant runtime for the tests during the evolution of the system. Only for version 12, the runtime exploded and grew even higher for version 13. Additionally, an exceptional test failed for version 12 and was fixed only in the following version. The question we followed further was: *What caused this increase of the runtime in version 12 and does it have a relation to the failed test case?*

The first investigation focused on the evolution of the metrics for the structural software artifacts. All observed values, which are the number of classes, number of methods, number of test cases and number of changes, grew continuously over the several versions of the library. None of the metrics



(a) Test run results, no errors occured.

(b) Structural elements.

Figure 4.3: Evolution of Hessian over 13 Versions. (*10) denotes values that have been scaled down by a factor of 10.

revealed any unconventional peaks. The schematic evolution of the corresponding values can be seen in Figure 4.3(b).

We continued with a detailed analysis of the runtime behavior for all the changes performed after the eleventh version. For that purpose, we ran the system for every captured Changebox of the twelfth and thirteenth version. This corresponds to regression testing the system after every single change, only performed retrospectively. The detailed results are shown in Figure 4.4.

The script to generate the values in Figure 4.4 is very simple and straightforward. It only took the summed time of all test runs to execute. No pre-computations were required. The following code shows how the runtime was determined. `versions11to13` is a list of all Changeboxes from version 11 to 13, which was derived from the directed graph of Changeboxes.

```
versions11to13 collect: [ :cbx |
    CurrentScope use: cbx during: [
        [ (PackageInfo named: 'Hessian') allTestCases
            run: TestResult new ] timeToRun ] ]
```

Using this method, we were able to measure the direct impact of every single change. The runtime remained mostly constant during the development of version 12. Towards the end of the iteration, two changes doubled the runtime of the 105 tests present. The method in question is the method `#convert:` of the `HSDecodingContext` that converts a UTF-8 string into a wide string, which is a fairly expensive function. This method is called by the modified version of `#decodeShortString` and `#decodeLongString`, which caused the major runtime increase. The failed test decoded a previously encoded string and asserted that the result is equal to the source. Obviously,

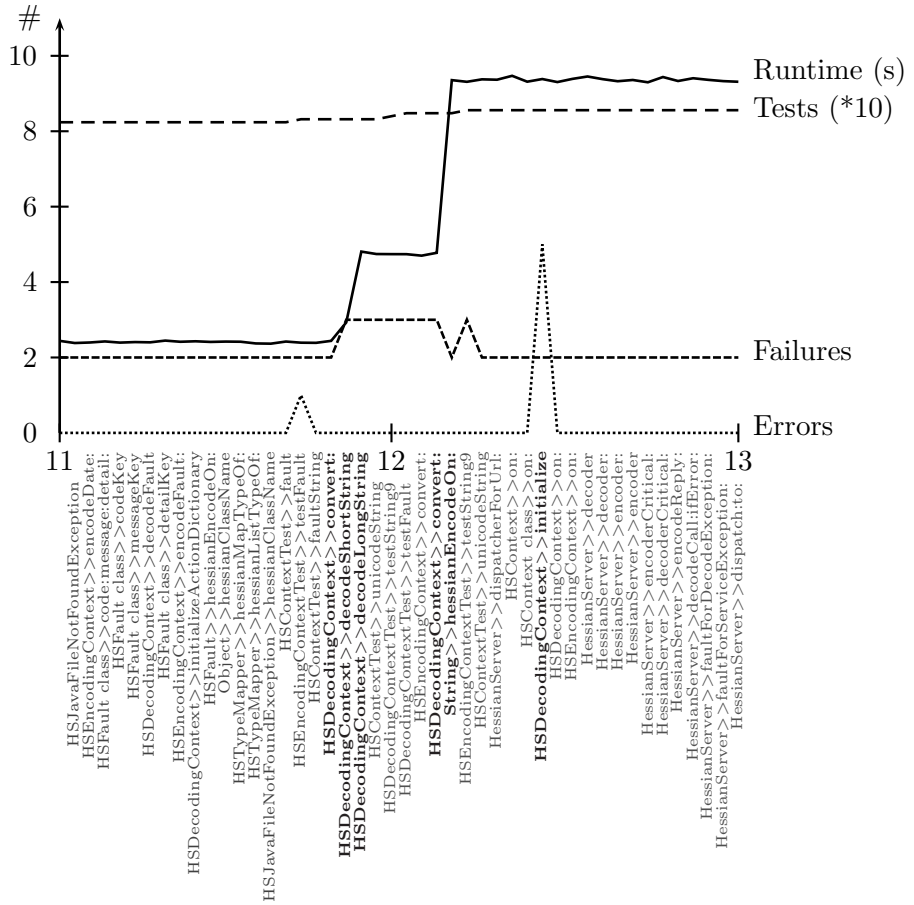


Figure 4.4: Hessian test runs for every Changebox of versions 12 and 13.

the new feature for decoding UTF-8 strings did not yet have a counterpart that correctly encoded a UTF-8 string.

The elimination of this exceptional failure was addressed at the beginning of the development for version 13. The `#convert:` methods of both `HSDecodingContext` and `HSEncodingContext` were refactored. The modification of the `String` extension `#hessianEncodeOn:` to encode a string to UTF-8 (using the `#convert:` methods) could resolve the failure, but, being as expensive as decoding, doubled the runtime of the tests a second time.

In the following, another test was created to reflect the new encoding. It first failed, but was immediately fixed by an update of a helper method of that test. During the remaining changes, the runtime stayed constantly at the higher value, being about four times slower than before the UTF-8 conversion was introduced.

Figure 4.4 illustrates another property of Changeboxes. The execution scope of a Changebox does not necessarily have to represent a consistent system. A change might introduce many errors to a system, as did the definition of the `HSDecodingContext>>initialize` method. It is not unusual that changes break a system, and there is no reason to not encapsulate them since they hold own semantics as well. Changeboxes allow *views of possibly inconsistent systems*.

With this evolution analysis, we could show that adding UTF-8 conversion to the Hessian protocol resulted in a dramatic decrease of the performance for the test cases. An additional failure emerged in one version because the decoding function was completed there, but the encoding function only in the subsequent version.

Determining the test run values for every single change was very easy by using Changeboxes. Because the complete history of a system is available, no tedious version switching was required to run the tests. Having every single change runnable allowed us to determine their impact on the execution time and on the test results and thus to directly relate changes to exceptional values of these metrics. This detailed identification of the corresponding methods would not have been possible otherwise without a profound knowledge of the system. We believe that Changeboxes greatly ease analysis of the evolution of software systems and also provide many new possibilities for this area.

Chapter 5

Conclusions

This chapter concludes the thesis by discussing the most important concepts of Changeboxes (Section 5.1) and the issues not covered yet by our implementation (Section 5.2). We state our main contributions to the field of software change (Section 5.3) and discuss the work future investigations on Changeboxes should tackle (Section 5.4).

5.1 Discussion

With Changeboxes, it is possible to explore different runnable versions of a software system simultaneously. The complete history of changes remains available, because once created, a Changebox is immutable. This guarantees unlimited navigation in time: each version that ever existed can be returned to. There is no need for undo or redo functions anymore since it is possible to go to any captured snapshot of the system, without losing others. Every Changebox can be extended by creating new Changeboxes. Having the entire history of changes provides new kinds of data that helps one to understand how systems evolve. Various reengineering efforts, for example, invest a lot of work into recovering the meaning of changes (*e.g.*, by identifying refactorings [Deme00, Gorg05]). With Changeboxes, the semantics are directly available and we can concentrate on other questions, like on understanding of why changes happened. Furthermore, any additional information could be encapsulated in Changeboxes, like clicks, menu activations or keys pressed to perform a change. Such information would reveal data about which tools are used by programmers and how they are used.

Changeboxes encapsulate the process of change using change specifications that work on software elements. Both elements and change specifications are open for extensions, so that any kind of change on software artifacts may

be modeled. Changeboxes may be used to package bug fixes, refactorings or new program features.

Elements are either primitive — representing a runnable instance, *e.g.*, a compiled method — or non-primitive — representing an artifact held by a runnable instance, *e.g.*, an instance variable specified by a class. The advantage of this separation lies in the additional semantic information that can be captured. Consider the example where an instance variable is added to a class, which causes several other primitive elements to be modified in the class rebuilding process (*i.e.*, all methods of this class and its subclasses are recompiled to reflect the new instance variable offset). Having a dedicated element for the instance variable allows one to create a change specification that solely represents the definition of an instance variable. The additional low-level modifications (*i.e.*, the recompilation of all methods) depend on the implementation of a language and do not hold additional semantic information (*i.e.*, the method definitions are not changed). Therefore, the semantics of the change are completely covered by the definition specification of the instance variable, while the resulting runnable entities are all captured by the corresponding Changebox. Would this specific change, on the other hand, be specified for a class element, any other fields besides the defined instance variable would form a part of the corresponding specification as well. The specification would therefore explicitly define unaffected artifacts and hold misleading change information. In general, each software artifact of interest should be modeled with a separate element in order to collect semantic information about its changes. In our implementation, we confined ourselves to modeling the most important ones.

Change specifications describe the process of how certain elements in one system snapshot are changed to yield a new snapshot. The possible transformations and their corresponding implementations depend on the programming language and are not limited to simple changes like definition or removal. High-level change specifications bring the advantage of more appropriate semantic information and are kept together with the results of their related low-level edits in one Changebox. By capturing high-level changes directly in the tools that produce them, the purpose rather than simply the effects of a change can be encapsulated. In this way, change specifications (*e.g.*, for refactorings) can be applied generically to arbitrary snapshots, taking the respective existing elements into account. A method renaming specification, for example, can be applied to different snapshots and also update all senders of the renamed method there. This may be interesting to ease framework refactorings. All refactoring transformations would be stored in change specifications. The specifications are then available to clients that want to migrate to a new framework version. By applying the specifications to their own system, the code is transformed appropriately.

Agile environments require the developers to integrate their changes frequently into the versioning system to stay as much in contact with the overall development as possible. The possibilities for collaboration offered by current versioning systems entail a programmer to checkout a snapshot from the system, perform changes locally and then commit them. For a Changebox-aware system, we can envision a single server as development platform where programmers transparently and concurrently perform changes. They can directly see who is changing the system when and where, and can choose to integrate at any point. Because all changes are on-line, the system does not have to be shut down and restarted again, allowing Changeboxes to be deployed dynamically. The execution scope may be dynamically switched with Changeboxes. As such, Changeboxes serve as a general-purpose mechanism to control the scope of software evolution.

This can be particularly useful for systems that have to be running virtually all the time. Upgrading a system to a new version may happen at runtime, affecting existing objects directly. With the execution scope attached to a user session, it is possible that existing sessions continue with the old versions, while new ones begin with the upgraded system. Or only certain users may be granted a preview of the upgrade before it is released to the broad public. Of course, scope-aware sessions are not limited to subsequent versions, but can also be used to completely personalize a system. In collaborative work applications, for example, objects could exhibit different behavior or properties depending on who is interacting with them. For each user, a Changebox providing the corresponding active features may be specified [Goke02]. With Changeboxes, applications can dynamically adapt their behavior at runtime. Having fine-grained control over the scope of the evolution and the runtime behavior of a system creates numerous new possibilities for product line management.

Context aware systems (*e.g.*, context-oriented or aspect-oriented programming languages) allow for different simultaneous views of their software artifacts as well. While techniques like before and after methods [Cost05] are not directly represented with Changeboxes yet, the foundation of different concurrently active element versions is existing. Based on the execution scope given by a Changebox, a context may be defined. Before and after change specifications, for example, could be implemented to concatenate methods in an appropriate way. Layers may be expressed directly with different Changeboxes and be merged for execution. Various context aware systems might build on top of Changeboxes, which provide a unifying approach to model change.

By modeling classboxes, we could show that Changeboxes can be used to package extensions of existing components. Conventional class extensions (as, *e.g.*, in Smalltalk) add or override methods of existing classes. Because

these extensions are global, they might break existing clients. With Changeboxes, extensions can be encapsulated and are only visible in a well-defined scope. The property of Changeboxes that different versions of the same software artifact can exist concurrently may thus be exploited in temporal and spatial dimensions.

In our implementation, not the complete system is encapsulated in Changeboxes, but only changes explicitly captured in a work session. If no variants are found in the ancestry of the Changebox defining the current execution scope, we fall back to the artifacts present in the global scope. Like this, Changeboxes may be used only for certain parts of a project, while the other components behave as usual.

5.2 Open Issues

A few issues remain unsolved in our implementation of Changeboxes. Some already have possible workarounds and others do not affect a system significantly, but they should all be tackled for a complete model:

- *Migrating instances between different scopes.* A major shortcoming is that instances always stick to the version of their class from the execution scope they were created in. The need for migrating instances to move them between different scopes is discussed in Section 5.2.1.
- *Changeboxes for system components.* System components are currently excluded from being encapsulated in Changeboxes. The reasons and a possible solution for this problem are presented in Section 5.2.2.
- *Fixed class references* Because of the reflective nature of Smalltalk, class references may still be fixed in some cases and evade the dispatch mechanism. Certain workarounds exist and are discussed in Section 5.2.3.

5.2.1 Migrating Instances between Different Scopes

Every object knows the class it was instantiated from, namely the exact version (*i.e.*, instance) of that class. Whenever the shape of a class changes, for example when instance variables are modified or the format of the class changes, all instances have to be migrated to the new version. The conventional class modification procedures in Smalltalk offer a rudimentary migration of instances. Newly added fields are simply initialized with a `nil` value. This migration affects all instances in the system, independent of any specified execution scope.

When a certain class is modified in a special execution scope, this could result in undefined behavior. Therefore, the basic migration was deactivated in our implementation. Instances keep the shape of the class in the execution scope they were created in. Currently, Changeboxes do not offer any mechanism that would enable the migration to new class shapes.

Support for migrating instances between different execution scopes is necessary because objects may be accessed from the scope of different Changeboxes. A sophisticated solution could, for example, encapsulate migration scripts in Changeboxes which would upgrade instances according to the performed class shape change (eg, they could initialize newly added instance variables). These scripts could be executed whenever the scope is switched.

In Smalltalk, classes have initialization and unloading methods, *i.e.*, methods that are invoked from the system when a class is newly created or removed. With Changeboxes, where the visible software artifacts can be dynamically switched together with the execution scope, the semantics of these callback methods change. Classes are not globally loaded or removed from a system, they are only visible or hidden in different scopes. Like for migrating objects, a solution has to be found that manages the circumstances when classes are introduced or excluded when switching the execution scope. In the current implementation, unloading methods are not executed when a class is removed in a work session in order to prevent undefined behavior.

5.2.2 Changeboxes for System Components

In our implementation, the Changebox defining the execution scope for a block closure has to be set explicitly. This introduces several restrictions for the use of Changeboxes. The most important one is that certain Smalltalk kernel components, Changebox-aware development tools, and the Changebox packages themselves cannot benefit from Changeboxes. The development process of a system relies highly on the functionality of these components. The possibility to further evolve them in an insulated scope would be a big advantage. Because the execution scope is specified for a complete block of code, it is not possible (or only with a disproportional effort) to separate the scope of a modified software artifact from the scope of the components that perform the change. For example, changing a method of the compiler immediately falls back on this method in the compilation process. Potential errors would invalidate the compiler, resulting in a deadlocked system. With separate scopes, the working version of the compiler in one scope could modify an experimental version in another scope.

This problem could possibly be solved by implicitly specifying the execution scope based on the current execution context. Development processes could run in a scope separated from the artifact they are currently changing, even when this artifact is a part of such a process. Scope could, for example, be defined depending on where a call originates from, distinguishing the execution of reflective system methods (*e.g.*, to retrieve information about the artifact to change) from inner-component calls (*e.g.*, messages sent within the compiler).

For other domains, implicit scope specification might be interesting as well, for example for context-aware, mobile applications.

5.2.3 Fixed Class References

To allow classes being dispatched depending on the current execution scope, the compiler was modified to replace class references with a lookup mechanism. This works well for references in source code, but as Smalltalk is a highly reflective language in which classes are ordinary objects, they can be stored in variables as well. Because variables store a direct reference to an object, reading such a variable results in the class from the execution scope where it was originally stored instead of where it is read. For the moment, this issue can be worked around by only storing the name of a class in a variable and performing a lazy lookup (*e.g.*, in an accessor method) whenever the actual class object is needed.

Another issue resulting from the compiler modification is that the class lookup is only introduced for newly compiled methods. System library methods still contain direct class references and would need to be recompiled to perform a lookup. Because such classes are currently not affected by Changeboxes and therefore do not need to be dispatched, we abstained from recompiling these libraries. If required, this can be done quickly.

5.3 Contributions

In this thesis, we proposed Changeboxes as a unifying mechanism for modeling software change. To comply with the importance of change, we represent it as a first-class entity. Changeboxes allow one to exploit change for many different domains. In the following, we expose our key contributions.

Metamodel for Software Change. Changeboxes provide a generic metamodel for representing software change. They not only encapsulate the effects, but also the *semantics* of change using change specifications. Change-

boxes can model *temporal and spatial dimensions* of software change. Because Changeboxes are *immutable*, they may be arbitrarily and safely extended to form new ones. Changeboxes encapsulate the entire history of a software system, which creates many possibilities for potential applications managing or analyzing software evolution.

Capturing Changes. Changes are captured automatically at the level of the reflective language kernel and the integrated development environment (IDE). A *bottom-up* approach enables a fine-grained recording of basic changes like definition and removal of software artifacts, independent of where a change action originated. To capture the meaning of complex changes (*e.g.*, refactorings) that involve several basic change operations, a *top-down* approach hooks into the IDE, where most semantic information about such operations is available.

Different Active Execution Scopes. Every captured Changebox provides an *execution scope* that defines the active software artifacts constituting a system. Different scopes, *i.e.*, different versions of the same system, can be simultaneously active within a single running application. With the ability to change the execution scope dynamically, bug fixes or new features can be safely integrated into a running system without impacting active user sessions. Because several versions of the same software artifact may coexist, Changeboxes may be used to model dynamic module systems and other spatial concepts of change. The evolution of a system may be studied based on fine-grained snapshots which are all directly runnable.

Integration into Programming Environment. Changeboxes are tightly integrated into the development environment. All conventional programming tools such as code browsers, debugger, test runner and source control facilities are made Changebox-aware. This allows a developer to work conventionally without the need to learn handling new applications. Different windows may represent different views of a system, *i.e.*, they may display or execute code in the scope of different Changeboxes concurrently. A tool to manage work sessions acts as a central hub for developing in several branches and for switching scope.

5.4 Future Work

Changeboxes offer a simple and uniform mechanism to encapsulate changes. With our proof-of-concept implementation, we developed a runnable, Changebox-aware environment that may be used for a multitude of experiments.

Besides the issues mentioned in Section 5.2, further investigations in this topic should address (i) performance issues by considering an implementation at the virtual machine level and (ii) a refinement of how and which changes are encapsulated in order to gain additional semantic information. These two issues are discussed in the two concluding sections.

5.4.1 Performance Enhancements

One problem for our implementation is that Smalltalk does not provide a notion of execution scope. Scope must be simulated by modifying the method lookup and introducing a dynamic class reference resolution. The performance analysis in Section 4.1.2 showed that most of the time needed for a method or class lookup is spent in image level Smalltalk code.

By implementing the lookup mechanism at the virtual machine level, a significant speed improvement should be possible. With this modification, execution scope would become an implicit property of the language. This should also ease the introduction of Changeboxes for the entire system, including the kernel libraries.

Furthermore, the caching logic could be improved as well to provide an additional speed-up. The current implementation is very rudimentary and did not undergo any optimization process. For example, one cache per message call site in relation to the current execution scope, similar to a polymorphic inline cache (PIC) [Holz91] could bring some enhancements.

Not only speed, but also space can become a critical factor. The growth of memory is currently linear to the number of Changeboxes in the system. All Changeboxes captured are kept in the image, even when they contain changes overridden long ago. A strategy to swap-out unused Changeboxes and flush the lookup caches would heavily reduce the memory footprint in real world usage.

5.4.2 Finer-grained Change Information

Changeboxes offer support for capturing changes of arbitrary software elements. Currently, only the most basic elements like classes, methods and fields are represented. Finer-grained elements (*e.g.*, statements, blocks or packages) are required for more detailed change information. The inclusion of newer concepts like Traits [Scha05] might be interesting as well.

Aside from simple definition and removal changes, there is only one change specification that encapsulates a more complex change: the renaming specification. To fully exploit the mechanism to capture high-level changes (as

described in Section 3.3.3), additional change specifications have to be modeled for the corresponding transformations. Especially, support for various well-known refactorings would be significant. A tight integration with Smalltalk's refactoring browser [Robe97] or the succeeding refactoring engine is possible. Composable change specifications could be helpful to model certain change processes, for example when several changes are initiated by a single action (*e.g.*, a method consisting of several changeable statements is saved).

Together with the introduction of additional elements and change specifications, the merge logic has to be evolved as well. The newly added entities have to be taken into account for the merge algorithm. Additional investigations into a sophisticated and more general merge process will be needed for that. The possibility of merging only selected parts from the ancestry of a Changebox might have to be examined (*e.g.*, to support explicit imports for classboxes). With new merge strategies, an even wider range of semantic meanings would arise for merging Changeboxes.

Appendix A

Short Guide to Our Implementation

A.1 Installing Changeboxes

Changeboxes are tested and developed with Squeak 3.9. You need an image that is compiled with NewCompiler[1] [NewC].

A ready to run demo image with Changeboxes is available at `http://www.iam.unibe.ch/~scg/Research/Changeboxes/ChangeboxesDemo.zip`.

A.1.1 Source Installation

All sources can be loaded from SqueakSource using Monticello [Brya].

1. Get the latest version of Squeak 3.9 and the virtual machine for your operating system at `http://ftp.squeak.org/3.9`
2. Get the latest version of *AST* at `http://www.squeaksource.com/AST`
3. Get the latest version of *NewCompiler* at `http://www.squeaksource.com/NewCompiler`
4. Compile the entire image using `Recompiler`:

   ```
   [ Recompiler new inspect; recompileImage ] forkAt: 30
   ```

5. Get the latest *ChangeBoxes* package at `http://www.squeaksource.com/ChangeBoxes`

[1] A precompiled image with NewCompiler is available at `http://www.iam.unibe.ch/~scg/Research/NewCompiler/NewCompiler.zip`. With this image, you can skip steps 2. to 4. in the following installation guide and directly start loading the Changebox packages.

6. Get the latest *ChangeBox-activation* package at the same place. This loading order is essential.

7. Run the SUnit Tests in the *ChangeBoxes-Tests* package in order to assert everything went well.

If you like syntax highlighting as you type in your own work sessions as well, you can get *Shout* like this:

1. Get *ShoutUsingNewCompiler.3.15-tween.73* at `http://www.squeaksource.com/shout`

2. Get *ShoutOmniBrowser-tween.3* at the same place

3. Get the latest version of *ChangeBox-shout* at `http://www.squeaksource.com/ChangeBoxes`

A.2 Getting Started

After loading Changeboxes, you may want to try the following small tutorial.

- First, you want to create a new work session, where all your changes will be put. For that purpose, open the Work Session Browser from your world menu. You should see a window like in Figure A.1 with three columns in the top and one big pane in the bottom, all empty at first. In the upper left pane, right-click and choose *create worksession*. In the dialog, enter a name for your work session and change the color if you like, then click on *accept*.

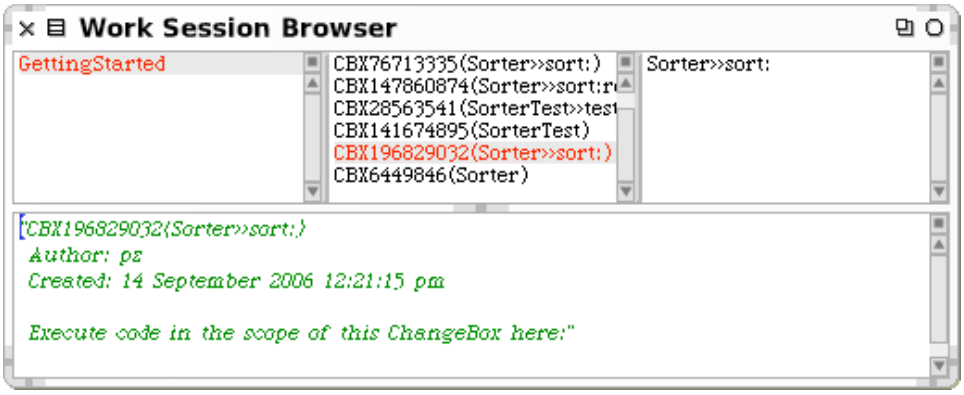


Figure A.1: The Work Session Browser with our project.

- There should now be an entry with the name of your created work session. On the context menu you will find the option *browse*, which will open a new image browser that is aware of your work session.
- There we go. All the modifications you perform in this browser will now be local to your work session. The rest of the system will not be aware of your changes at all. So let's start with some coding. Create a new category, e.g. *CBXGettingStarted*, and create a new class called `Sorter` therein. We will code a quicksort algorithm. Let's add the following method to the class `Sorter`:

```
sort: aCollection
  ^aCollection isEmpty
    ifTrue: [ aCollection species new ]
    ifFalse: [
      (self sort: (aCollection allButFirst
        select: [:each | each < aCollection first])) ,
      (aCollection species with: aCollection first) ,
      (self sort: (aCollection allButFirst
        select: [:each | each >= aCollection first])) ]
```

- Of course, we need to test that algorithm. So let's create a class `SorterTest` that extends `TestCase` and put the method `#testSort` in there (still in the browser for our work session):

```
testSort
  self assert: (Sorter new sort: #(2 4 3 6)) = #(2 3 4 6).
  self assert: (Sorter new sort: #(2 3 4 3 2)) = #(2 2 3 3 4).
  self assert: (Sorter new sort: #(1)) = #(1).
```

- If you open a normal image browser from the open menu, you will recognize that neither the class `Sorter` nor the class `SorterTest` are existing there. Your changes are only visible in the browser with the assigned work session. (Caveat: This is only true for `OBSystemBrowser`. The old browser will still display both of the classes, as it is not aware of Changeboxes). So let's see what we did so far. For that purpose, go back to the Work Session Browser. You should now see a list on the right with all the changes you performed so far.
- To run your test, go to the top Changebox in the center column of the Work Session Browser and select *open test runner* from the context menu. This will open a Test Runner that is aware of the selected Changebox and its execution scope. Maybe you need to refresh the categories list in the Test Runner in order to see your category containing `SorterTest`. Select only this class, run the test and enjoy the green light if everything went right.
- Our code for the sorting algorithm is not that beautiful yet. The two calls for sorting the rest of the list could be put in their own method:

```
sort: aCollection relation: aSelector
  ^self sort: (aCollection allButFirst select: [:each |
    each perform: aSelector withArguments: aCollection first])
```

- So the `#sort:` method can be changed to the following:

```
sort: aCollection
  ^aCollection isEmpty
    ifTrue: [aCollection species new]
    ifFalse: [
      (self sort: aCollection relation: #<) ,
      (aCollection species with: aCollection first) ,
      (self sort: aCollection relation: #>=) ]
```

- Let's test. Switch to the Test Runner and run again. Still green. But wait a minute... What did we actually test? Not our new changes, no, but the snapshot of the system we had before. The Test Runner is still executing its tests in the scope of the previous Changebox. Nobody told it to switch to the new execution scope. This also shows that the code before the refactoring is still completely runnable, no matter what we changed. So before running the tests on the new version, update the Test Runner by opening it from the last Changebox in the Work Session Browser. For more convenience, you could also open the Test Runner on the work session, so the tests will always run on the latest snapshot. What's the result of running the tests now? Red. By clicking on the failed test, you get a Debugger saying that instances of `SmallInteger` are not indexable. What happened? A small research shows that we used the first element of our collection as a parameter for `#perform:withArguments:`, instead of an `Array` containing that element. The fix is (change in your work session aware browser):

```
sort: aCollection relation: aSelectors
  ^self sort: (aCollection allButFirst select: [:each |
    each perform: aSelector
         withArguments: (Array with: aCollection first)])
```

- If you now update your Test Runner again and run the test, everything should be green. Congratulations!

- Let's do some reverse engineering at the end: Open an inspector for your work session from the context menu in the Work Session Browser. Then evaluate the following code in the inspector window:

```
| cursor breakingChange |
breakingChange := nil.
cursor := self changebox.
[ breakingChange isNil ] whileTrue: [
  cursor doScoped: [
```

```
    [(Smalltalk at: #SorterTest) new testSort]
      on: Error
      do: [ breakingChange := cursor ] ].
    cursor := cursor ancestor ].
breakingChange specifications first
    definitionString asString inspect.
```

- What you get is the source code of the last change that broke the tests, a change of `Sorter>>sort:`. You see that the change is the refactored `#sort:` that uses the new `#sort:relation:` method.

  Of course, here we did not change anything else, but imagine a big system with a lot of parallel changes, where iterating through all versions and executing them directly could reveal many important details.

We hope that these few steps helped you to get more familiar with Changeboxes. Another approach to take is to create a few more work sessions and play around a little bit. `CBXMethodChangeTest` and `CBXClassChangeTest` are also good entry points to directly see the reflective kernel functions.

Appendix B

An Introduction to Squeak

Changeboxes were developed in Squeak [Inga97], an open-source dialect based on Smalltalk-80 [Gold83]. Smalltalk is a dynamically typed, object-oriented programming language from top to bottom. In Smalltalk, everything is an object, *i.e.*, an instance of a corresponding class that describes properties and behavior. The language is built as a powerful reflective system, implemented in itself. The classes and methods that constitute a system are Smalltalk objects themselves.

A Squeak system consists of two major components: the *virtual machine* and the *image*. The virtual machine provides an environment to allocate memory space, execute bytecode and a set of additional primitive calls as an interface to the underlying operating system. The Smalltalk image contains the entire structure and state of the system, both represented by congeneric objects. There is no separation of program code defined by classes and methods and the runtime objects this code instantiates.

The reflective capabilities and the availability of the complete system libraries in the image enables the modification of nearly everything in Smalltalk from within a running program. Methods can be added and even modified for any object in the image, including the kernel classes that define the system. The development environment is part of the image as well and is equally open for modifications. The class building and method compilation processes are both implemented using Smalltalk objects.

In Smalltalk, basically two classes encode the runnable parts of an object oriented software system: `Class` defines classes and `CompiledMethod` represents methods. The instances of these two classes build the structure of a running system. `Class` objects provide the information for the virtual machine to instantiate new objects and `CompiledMethod`s contain executable instructions in the form of bytecode. Developing a software system

in Smalltalk mainly involves the permanent creation, modification and removal of instances of these two objects. In most cases, these changes are performed in an abstracted way over an appropriate IDE.

The reflective system with its wide possibilities formed a crucial part in the development of Changeboxes. Modifications of essential system objects were feasible in Smalltalk so that their behavior could be adapted to our requirements. Changes to software artifacts could be captured by intercepting system methods and the scope for executing code could be defined in a general way.

To ease portability, deployment and maintenance, Changeboxes are completely implemented using the facilities provided inside the Squeak image. No changes to the virtual machine have been conducted.

Index

List of Figures

List of Tables

Bibliography

[Ache00] Franz Achermann and Oscar Nierstrasz. "Explicit Namespaces". In: Jürg Gutknecht and Wolfgang Weck, Eds., *Modular Programming Languages, Proceedings of JMLC 2000 (Joint Modular Languages Conference)*, pp. 77–89, Springer-Verlag, Zürich, Switzerland, Sep. 2000.

[Ache01a] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. "Piccola — a Small Composition Language". In: Howard Bowman and John Derrick, Eds., *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pp. 403–426, Cambridge University Press, 2001.

[Ache01b] Franz Achermann and Oscar Nierstrasz. "Applications = Components + Scripts — A Tour of Piccola". In: Mehmet Aksit, Ed., *Software Architectures and Component Technology*, pp. 261–292, Kluwer, 2001.

[Aldr04] Jonathan Aldrich. "Open Modules: Reconciling Extensibility and Information Hiding". In: Lodewijk Bergmans, Kris Gybels, Peri Tarr, and Erik Ernst, Eds., *SPLAT: Software engineering Properties of Languages for Aspect*, March 2004.

[Aldr05] Jonathan Aldrich. "Open Modules: Modular Reasoning About Advice". In: *Proceedings ECOOP 2005*, pp. 144–168, Springer Verlag, Glasgow, UK, July 2005.

[Arms03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology Stockholm, 2003.

[Arms96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[AST] "AST, an Abstract Syntax Tree package for Squeak". http://www.squeaksource.com/AST.html.

[Atki98] David L. Atkins. "Version Sensitive Editing: Change History as a Programming Tool". In: *System Configuration Management: ECOOP'98 SCM-8 Symposium*, pp. 146–157, Springer Verlag, Brussels, Belgium, July 1998.

[Beck98] Kent Beck and Erich Gamma. "Test Infected: Programmers Love Writing Tests". *Java Report*, Vol. 3, No. 7, pp. 51–56, 1998.

[Berg03a] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. "The Classbox Module System". In: *Proceedings of the ECOOP '03 Workshop on Object-oriented Language Engineering for the Post-Java Era*, July 2003.

[Berg03b] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. "Classboxes: A Minimal Module Model Supporting Local Rebinding". In: *Proceedings of Joint Modular Languages Conference (JMLC'03)*, pp. 122–131, Springer-Verlag, 2003.

[Berg05a] Alexandre Bergel. *Classboxes — Controlling Visibility of Class Extensions*. PhD thesis, University of Berne, Nov. 2005.

[Berg05b] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. "Classboxes: Controlling Visibility of Class Extensions". *Computer Languages, Systems and Structures*, Vol. 31, No. 3-4, pp. 107–126, Dec. 2005.

[Berl90] Brian Berliner. "CVS II: Parallelizing Software Development". *Proc. The Advanced Computing Systems Professional and Technical Association (USENIX) Conf.*, pp. 22–26, 1990.

[Bobr80] Daniel G. Bobrow and Ira P. Goldstein. "Representing Design Alternatives". In: *Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior*, July 1980.

[Bric05] J. Brichau and M. Haupt. "Survey of Aspect-oriented Languages and Execution Models". Tech. Rep., AOSD-Europe-VUB-01, May 2005.

[Brya] Avi Bryant. "Monticello". http://www.wiresong.ca/Monticello.

[Char06] Anis Charfi, Michel Riveill, Mireille Blay-Fornarino, and Anne-Marie Pinna-Dery. "Transparent and Dynamic Aspect Composition". In: *In Proceedings of the 4th Software Engineering Properties of Languages and Aspect Technologies (SPLAT) Workshop*, March 2006.

[Coll04] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly & Associates, Inc., 2004.

[Conr98] Reidar Conradi and Bernhard Westfechtel. "Version Models for Software Configuration Management". *ACM Computing Surveys*, Vol. 30, No. 2, pp. 232–282, June 1998.

[Cost05] Pascal Costanza and Robert Hirschfeld. "Language Constructs for Context-oriented Programming: An Overview of ContextL". In: *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, ACM Press, New York, NY, USA, Oct. 2005.

[Deme00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. "Finding Refactorings via Change Metrics". In: *Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pp. 166–178, ACM Press, New York NY, 2000. Also appeared in ACM SIGPLAN Notices 35 (10).

[Denk06] Marcus Denker, Stéphane Ducasse, and Éric Tanter. "Runtime Bytecode Transformation for Smalltalk". *Journal of Computer Languages, Systems and Structures*, Vol. 32, No. 2-3, pp. 125–139, July 2006.

[Dig05a] Daniel Dig and Ralph Johnson. "The Role of Refactorings in API Evolution". In: *Proceedings of 21st International Conference on Software Maintenance (ICSM 2005)*, pp. 389–398, Sep. 2005.

[Dig05b] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. "Automatic Detection of Refactorings for Libraries and Frameworks". *Proc. sixth ECOOP Workshop on Object-Oriented Reengineering*, 7 2005.

[Dig06] Danny Dig and Ralph Johnson. "How do APIs evolve? A story of refactoring". *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, Vol. 18, No. 2, pp. 83–107, Apr. 2006.

[Duca99] Stéphane Ducasse. "Evaluating Message Passing Control Techniques in Smalltalk". *Journal of Object-Oriented Programming (JOOP)*, Vol. 12, No. 6, pp. 39–44, June 1999.

[Erns99] Erik Ernst. "Propagating Class and Method Combination". In: R. Guerraoui, Ed., *Proceedings ECOOP '99*, pp. 67–91, Springer-Verlag, Lisbon, Portugal, June 1999.

[Feat89] Martin S. Feather. "Detecting Interference when Merging Specification Evolutions". *Proc. Fifth International Workshop on Software Specification and Design*, pp. 169–176, 1989.

[Fowl99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Gall03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. "CVS Release History Data for Detecting Logical Couplings". In: *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pp. 13–23, IEEE Computer Society Press, Los Alamitos CA, 2003.

[Gamm95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[Gass98] M.L. Gassanenko. "Context-Oriented Programming". In: *euroForth'98*, Schloss Dagstuhl, Germany, apr 1998.

[Girb04a] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. "Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes". In: *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pp. 40–49, IEEE Computer Society, Los Alamitos CA, Sep. 2004.

[Girb04b] Tudor Gîrba and Michele Lanza. "Visualizing and Characterizing the Evolution of Class Hierarchies". 2004.

[Girb05a] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, Nov. 2005.

[Girb05b] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. "Characterizing the Evolution of Class Hierarchies". In: *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pp. 2–11, IEEE Computer Society, Los Alamitos CA, 2005.

[Goke02] Ayse Göker and Hans I. Myrhaug. "User context and Personalisation". In: *ECCBR Workshop on Case Based Reasoning and Personalisation*, Aberdeen, UK, 2002. invited paper.

[Gold80a] Ira P. Goldstein and Daniel G. Bobrow. "Descriptions for a Programming Environment". In: *Proceedings of the First Annual Conference of the National Association for Artificial Intelligence*, Aug. 1980.

[Gold80b] Ira P. Goldstein and Daniel G. Bobrow. "Extending Object-Oriented Programming in Smalltalk". In: *Proceedings of the Lisp Conference*, pp. 75–81, Aug. 1980.

[Gold80c] Ira P. Goldstein and Daniel G. Bobrow. "A Layered Approach to Software Design". Tech. Rep. CSL-80-5, Xerox PARC, Dec. 1980.

[Gold83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[Gorg05] Carsten Görg and Peter Weissgerber. "Detecting and Visualizing Refactorings from Software Archives". In: *Proceedings of IWPC (13th International Workshop on Program Comprehension*, pp. 205–214, IEEE CS Press, 2005.

[Guzd01] Mark Guzdial and Kim Rose. *Squeak — Open Personal Computing and Multimedia*. Prentice-Hall, 2001.

[Hear06] David Hearnden, Michael Lawley, and Kerry Raymond. "Incremental Model Transformation for the Evolution of Model-Driven Systems". In: *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, pp. 321–335, Springer-Verlag, Berlin, Germany, 2006.

[Henk05] Johannes Henkel and Amer Diwan. "CatchUp!: capturing and replaying refactorings to support API evolution". In: *Proceedings International Conference on Software Engineering (ICSE 2005)*, pp. 274–283, 2005.

[Hess] "Hessian, a Squeak implementation of the binary web service protocol". http://www.squeaksource.com/hessian.html.

[Hick05] Michael Hicks and Scott Nettles. "Dynamic software updating". *ACM Trans. Program. Lang. Syst.*, Vol. 27, No. 6, pp. 1049–1096, 2005.

[Holz91] Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches". In: P. America, Ed., *Proceedings ECOOP '91*, pp. 21–38, Springer-Verlag, Geneva, Switzerland, July 1991.

[Hunt76] James Hunt and Douglas McIlroy. "An Algorithm for Differential File Comparison". Tech. Rep. CSTR 41, Bell Laboratories, Murray Hill NJ, 1976.

[Inga97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, A Practical

Smalltalk Written in Itself". In: *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pp. 318–326, ACM Press, Nov. 1997.

[Keay03] Roger Keays and Andry Rakotonirainy. "Context-oriented programming". In: *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pp. 9–16, ACM Press, New York, NY, USA, 2003.

[Kend99] Elizabeth Kendall. "Role Model Design and Implementations with Aspect-Oriented Programming". In: *Proceedings of OOPSLA '99*, pp. 353–369, Nov. 1999.

[Kent02] Stuart Kent. "Model Driven Engineering". *Proc. Integrated Formal Methods: Third International Conference*, 2002.

[Kicz97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming". In: Mehmet Aksit and Satoshi Matsuoka, Eds., *Proceedings ECOOP '97*, pp. 220–242, Springer-Verlag, Jyvaskyla, Finland, June 1997.

[Koeh03] Jana Koehler, Rainer Hauser, Shubir Kapoor, Fred Y. Wu, and Santhosh Kumaran. "A Model-Driven Transformation Method". *Proc. Seventh IEEE International Conference on Enterprise Distributed Object Computing*, pp. 186–197, 2003.

[Lanz02] Michele Lanza and Stéphane Ducasse. "Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics". In: *Proceedings of Langages et Modèles à Objets (LMO'02)*, pp. 135–149, Lavoisier, Paris, 2002.

[Lehm80] Manny Lehman. "On Understanding Laws, Evolution and Conservation in the Large Program Life-Cycle". *J Sys and Software*, Vol. 1, No. 3, 1980.

[Lehm85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[Lehm96] Manny Lehman. "Laws of Software Evolution Revisited". In: *European Workshop on Software Process Technology*, pp. 108–124, Springer, Berlin, 1996.

[Lipp92] Ernst Lippe and Norbert van Oosterom. "Operation-based merging". In: *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pp. 78–87, ACM Press, New York, NY, USA, 1992.

[MacK03] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd., 2003.

[Mens02] Tom Mens. "A State-of-the-art Survey on Software Merging". *IEEE Transactions on Software Engineering*, Vol. 28, No. 5, pp. 449–462, May 2002.

[Mezi03] Mira Mezini and Klaus Ostermann. "Conquering aspects with Caesar". In: *Proceedings of the 2nd international conference on Aspect-oriented software development*, pp. 90–99, ACM Press, 2003.

[Mezi97] Mira Mezini. "Dynamic Object Evolution without Name Collisions". In: *Proceedings ECOOP '97*, pp. 190–219, Springer-Verlag, June 1997.

[Muns94] Jonathan P. Munson and Prasun Dewan. "A Flexible Object Merging Framework". *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pp. 231–242, 1994.

[NewC] "NewCompiler, a Squeak compiler based on ClosureCompiler". http://www.squeaksource.com/NewCompiler.html.

[Nguy05] Tien Nguyen, Ethan Munson, and John Boyland. "An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services". In: *Internationl Conference on Software Engineering (ICSE 2005)*, pp. 215–224, ACM Press, 2005.

[Nier04] Oscar Nierstrasz and Marcus Denker. "Supporting Software Change in the Programming Language". In: *OOPSLA Workshop on Revival of Dynamic Languages*, Oct. 2004.

[Nier05] 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, Eds., *Proceedings of Software Composition 2005*, pp. 1–13, LNCS 3628, 2005. Invited paper.

[Nier06a] Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, and Adrian Lienhard. "Analyzing, Capturing and Taming Software Change". In: *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.

[Nier06b] Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. "Flattening Traits". *Journal of Object Technology*, Vol. 5, No. 4, pp. 129–148, May 2006.

[Opdy92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[Orio04] Manuel Oriol. *An Approach to the Dynamic Evolution of Software Systems*. PhD thesis, Centre Universitaire d'Informatique, University of Geneva, Apr. 2004.

[Otis91] Allen Otis, Paul Butterworth, and Jacob Stein. "The GemStone Object Database Management Systems". *Communications of the ACM*, Vol. 34, No. 10, pp. 64–77, Oct. 1991.

[Penn87] D. Jason Penney and Jacob Stein. "Class Modification in the GemStone Object-Oriented DBMS". In: *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, pp. 111–117, Dec. 1987.

[Putn] Colin Putney. "OmniBrowser, an extensible browser framework for Smalltalk". http://www.wiresong.ca/OmniBrowser.

[Reng06] Lukas Renggli. *Magritte – Meta-Described Web Application Development*. Master's thesis, University of Bern, June 2006.

[Robb05] Romain Robbes and Michele Lanza. "Versioning Systems for Evolution Research". In: *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pp. 155–164, IEEE Computer Society, 2005.

[Robb06] Romain Robbes and Michele Lanza. "Change-based software evolution". In: *Proceedings of EVOL 2006 (1st International ERCIM Workshop on Challenges in Software Evolution)*, pp. 159–164, 2006.

[Robe97] Don Roberts, John Brant, and Ralph E. Johnson. "A Refactoring Tool for Smalltalk". *Theory and Practice of Object Systems (TAPOS)*, Vol. 3, No. 4, pp. 253–263, 1997.

[Roun05] David Roundy. "Darcs: Distributed Version Management in Haskell". In: *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pp. 1–4, ACM Press, New York, NY, USA, 2005.

[Sato04] Yoshiki Sato and Shigeru Chiba. "Negligent class loaders for software evolution". *Proc. ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2004.

[Scha05] Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, Feb. 2005.

[Send03] Shane Sendall and Wojtek Kozaczynski. "Model transformation: the heart and soul of model-driven software development". *IEEE Software*, Vol. 20, pp. 42–45, 2003.

[Smit96] Randall B. Smith and Dave Ungar. "A Simple and Unifying Approach to Subjective Objects". *TAPOS special issue on Subjectivity in Object-Oriented Systems*, Vol. 2, No. 3, pp. 161–178, 1996.

[Stey96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. "Reuse Contracts: Managing the Evolution of Reusable Assets". In: *Proceedings of OOPSLA '96 (International Conference on Object-Oriented Programming, Systems, Languages, and Applications)*, pp. 268–285, ACM Press, 1996.

[Tich85] Walter F. Tichy. "RCS - a system for version control". *Software Practice and Experience*, Vol. 15, No. 7, pp. 637–654, July 1985.

[Tich88] Walter Tichy. "Tools for Software Configuration Management". In: *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 1–20, 1988.

[Wool96] Bobby Woolf. "The Null Object Pattern". In: *Design Patterns, PLoP 1996*, Robert Allerton Park and Conference Center, University of Illinois at Urbana-Champaign, Monticello, Illinois, 1996.