Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN
Vakgroep Computerwetenschappen
Software Languages Lab

Supporting Integration Activities in Object-Oriented Applications

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Verónica Isabel Uquillas Gómez

Promotoren:	Prof. Dr. Theo D'Hondt
	Dr. Stéphane Ducasse
Co-Promotor:	Dr. Andy Kellens

Brussel, Oktober 2012

Print: Silhouet, Maldegem

©2012 Verónica Isabel Uquillas Gómez

2012 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5718 213 6
NUR 989
Legal deposit D/2012/11.161/149

Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique — École doctorale SPI Lille
UFR IEEA

Supporting Integration Activities in Object-Oriented Applications

THÈSE

présentée et soutenue publiquement le 4 octobre 2012

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Verónica Isabel UQUILLAS GÓMEZ

Composition du jury

Rapporteurs :	Michele LANZA	(Professeur - Université de Lugano)
	Jurgen VINJU	(Directeur de Recherce - Centrum Wiskunde & Informatica - Amsterdam)
Examinateurs :	Nicolas ANQUETIL	(Maître de Conférence – Université de Lille1)
	Viviane JONCKERS	(Professeur – Vrije Universiteit Brussel)
	Julia LAWALL	(Directrice de Recherche – INRIA Paris)
	Ann NOWE	(Professeur – Vrije Universiteit Brussel)
Directeurs de thèse :	Theo D'HONDT	(Professeur – Vrije Universiteit Brussel)
	Stéphane DUCASSE	(Directeur de Recherche – INRIA Lille)
Co-Directeur de thèse :	Andy KELLENS	(Chercheur – Vrije Universiteit Brussel)

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

Numéro d'ordre: 40882

Abstract

Modern software is built by teams of developers that work in a collaborative environment. The goal of this kind of development is that multiple developers can work in parallel. They can alter a set of shared artifacts and inspect and integrate the source code changes of other developers. For example, bug fixes, enhancements, new features or adaptations due to changing environment might be integrated into the system release.

At a technical level, a collaborative development process is supported by version control systems. Since these version control systems allow developers to work in their own branch, merging and integration have become an integral part of the development process. These systems use automatic and advanced merging techniques to help developers to merge their modifications in the development repositories. However, these techniques do not guarantee to have a functional system.

While the use of branching in the development process offers numerous advantages, the activity of merging and integrating changes is hampered by the lack of comprehensive support to assist developers in these activities. For example, the integration of changes can have an unexpected impact on the design or behavior of the system, leading to the introduction of subtle bugs. Furthermore, developers are not supported when integrating changes across branches (cherry picking), when dealing with branches that have diverged, when finding the dependencies between changes, or when assessing the potential impact of changes.

In this dissertation we present an approach that aims at alleviating these problems by providing developers and, more precisely, integrators with semi-automated support for assisted integration within a branch and across branches. We focus on helping integrators with their information needs when understanding and integrating changes by means of characterizations of changes and streams of changes (*i.e.,* sequence of successive changes within a branch) together with their dependencies.

These characterizations rely on the first-class representation of systems' histories and changes based on program entities and their relationships rather than on files and text. For this, we provide a family of meta-models (*Ring*, *RingH*, *RingS* and *RingC*) that offer us the representation of program entities, systems' histories, changes and their dependencies, along with analyses for version comparison, and change and dependency identification. Instances of these meta-models are then used by our proposed tool support to enable integrators to analyze the characterizations and changes. *Torch*, a visual tool, and *JET*, a set of tools, actually provide the information needs to assist integration within a branch and across branches by means of the characterization of changes and streams of changes respectively.

Keywords: object-oriented programming; meta-models; history and version of programs; visualization; semantic merging; program analyses

Samenvatting

Hedendaagse software is het resultaat van een collaboratief ontwikkelingsproces met meerdere teams van ontwikkelaars. Het doel van dit proces is om het toe te laten dat ontwikkelaars gelijktijdig en onafhankelijk van elkaar kunnen werken. Hiervoor hebben ze toegang tot een gedeelde verzameling van artefacten die ze kunnen aanpassen, en hebben ze de mogelijkheid om de aanpassingen die andere ontwikkelaars maken aan de broncode te inspecteren en te integreren. Zo kunnen bijvoorbeeld *bug fixes*, verbeteringen en nieuwe functionaliteit tijdig geïntegreerd worden in een versie van een softwaresysteem.

Op een technisch niveau wordt dit collaboratief ontwikkelingsproces ondersteund door versiecontrolesystemen. Gezien deze versiecontrolesystemen het mogelijk maken voor ontwikkelaars om in hun eigen *branch* van het systeem te werken, zijn *merging* en integratie een volwaardig onderdeel van het ontwikkelingsproces geworden. Hiertoe bieden deze versiecontrolesystemen geavanceerde en geautomatiseerde merge-technieken aan die ontwikkelaars helpen om hun aanpassingen samen te voegen met de aanpassingen van andere ontwikkelaars. Echter, deze technieken garanderen niet dat het resultaat van dit samenvoegen tot een werkend systeem zal leiden.

Alhoewel het gebruik van *branching* binnen het ontwikkelingsproces vele voordelen biedt, worden de hieraan verbonden taken van het invoegen en integreren van aanpassingen bemoeilijkt door een gebrek aan ondersteuning. Bijvoorbeeld, het integreren van aanpassingen kan een onverwachte impact hebben op het ontwerp of het gedrag van het systeem, wat dan weer kan leiden tot de introductie van subtiele fouten. Bovendien wordt er aan ontwikkelaars geen ondersteuning geboden bij het integreren van veranderen die afkomstig zijn uit een andere *branch* van het systeem (het zogenaamde *cherry picking*), bij divergerende *branches*, bij het zoeken naar afhankelijkheden tussen aanpassingen, of bij het inschatten van de mogelijke impact van een verzameling veranderingen op het systeem.

In dit proefschrift stellen we een techniek voor die bovenvermelde problemen aanpakt door ontwikkelaars – en in het bijzonder integrators – semi-automatisch te assisteren bij het integreren van aanpassingen, zowel binnen één *branch* als tussen verschillende *branches*. We leggen hierbij de klemtoon op het helpen van integrators om de informatie te verkrijgen die ze nodig hebben om aanpassingen aan de software te begrijpen en te integreren. Hiervoor maken we gebruik van een karakterisering van aanpassingen en van aanpassingsstromen (dit zijn een opeenvolging van aanpassingen binnen een *branch*), te samen met een karakterisatie van de afhankelijkheden tussen de aanpassingen.

Deze karakteriseringen zijn gebaseerd op een eersterangs voorstelling van de historiek van een systeem en de aanpassingen die binnen deze historiek werden uitgevoerd. Deze voorstelling is gedefinieerd in termen van de feitelijke programma-entiteiten, in plaats van bestanden en tekst die integrators niet de noodzakelijke informatie verschaffen. Hiervoor bieden we een familie van meta-modellen aan (*Ring*, *RingH*, *RingS* en *RingC*) die een implementatie verschaffen van de voorstelling van programma-entiteiten, de historiek van het systeem, aanpassingen, en de afhankelijkheden tussen aanpassingen. Deze meta-modellen bieden ook de analyses aan om versies van een systeem te vergelijken, en om aanpassingen en afhankelijkheden te berekenen. Verder stellen we *tools* voor die, gebruik makende van instanties van onze meta-modellen, het mogelijk maken voor integrators om de karakteriseringen van aanpassingen te analyseren. De visuele *tool Torch* en de verzameling van *JET-tools*, voorzien in de informatie die noodzakelijk is om assistentie te bieden bij respectievelijk het integreren van aanpassingen binnen één *branch* en tussen verschillende *branches*.

Trefwoorden: objectgericht programmeren; meta-modellen; historiek en versies van programma's; visualisatie; semantisch *mergen*; programma-analyses

Résumé

De plus en plus de logiciels sont développés par des équipes de développeurs travaillant de manière collaborative en parallèle. Les développeurs peuvent altérer un ensemble d'artéfacts, inspecter et intégrer le code de changements faits par d'autres développeurs. Par exemple, les corrections d'erreurs, les améliorations ou nouvelles fonctionnalités doivent être intégrées dans la version finale d'un logiciel et ceci à différents moments du cycle de développement.

A un niveau technique, le processus de développement collaboratif est mis en pratique à l'aide d'outils de contrôle de versions (ex: git, SVN). Ces outils permettent aux développeurs de créer leurs propres branches de développement, faisant des tâches de fusion ou d'intégration de ces branches une partie intégrante du processus de développement. Les systèmes de versions de contrôle utilisent des algorithmes de fusion pour aider les développeurs à fusionner les modifications de leur branche dans le base de code commune. Cependant ces techniques travaillent à un niveau lexical, et elles ne garantissent pas que le système résultant soit fonctionnel.

Alors que l'utilisation de branches offre de nombreux avantages, la fusion et l'intégration de modifications d'une branche sur une autre est difficile à mettre en oeuvre du fait du manque de support pour assister les développeurs dans la compréhension d'un changement et de son impact. Par exemple, l'intégration d'un changement peut parfois avoir un effet inattendu sur le système et son comportement menant à des bugs subtiles. De plus, les développeurs ne sont pas aidés lors de l'évaluation de l'impact d'un changement, ou lors de la sélection de changements à intégrer d'une branche vers une autre (cherry picking), en particulier lorsque ces branches ont divergé.

Dans cette dissertation, nous présentons une approche dont le but est d'apporter des solutions à ces problèmes pour les développeurs, et plus précisément les intégrateurs. Cette approche se base sur des outils et solutions semi-automatisés aidant à de changements la compréhension à l'intérieur d'une branche ou entre branches. Nous nous attachons à satisfaire les besoins en information des intégrateurs quand ils doivent comprendre et intégrer des changements. Pour cela, nous caractérisons les changements et/ou séquences de changements et leurs dépendances.

Ces caractérisations sont basées sur la représentation comme citoyens de première classe de l'historique du système et des changements approtés considérant les entités logicielles (ex: classes ou méthodes) et leurs relations plutôt que des fichiers et du texte comme le font les outils de contrôle de versions. Pour cela, nous proposons une famille de méta-modèles (Ring, RingH, RingS et RingC) qui offrent une représentation des entités du système, de son historique, des changements apportés dans les différentes branches et de leurs dépendances. Des instances de ces meta-modèles sont ensuite utilisées par nos outils destinée à assister les intégrateurs: Torch, un outil visuel qui caractérise les changements, et JET un ensemble d'outils qui permettent de naviguer dans des séquences de changements.

Mots clés: programmation à objets; méta-modèles; historique et versions de programmes; visualisation de programmes; fusion sémantique; analyse de programmes.

Acknowledgments

This has been a long journey that has made a dream come true and that has provided me with many experiences that will be part of my memories for the rest of my life. Five years ago, I left Guayaquil – a city where the summer never ends in my home country Ecuador – to obtain a master degree in mostly cold and rainy Belgium.

After graduating, I was awarded a four-year Dehousse scholarship to pursue a PhD at the Vrije Universiteit Brussel. I am very grateful that I got this opportunity. Now, four years later I am writing this to express my gratefulness to everybody that contributed in some way to the document you are now holding and that helped me during the whole process of becoming a Doctor in Sciences.

First of all I would like to thank my promotors who were the masterminds behind my work. I deeply thank Theo D'Hondt, my promotor at the Software Languages Lab, who had to read my thesis in bumping buses while on holidays in Australia :) Thank you Theo because you guided me during my master and PhD, and you advised me while taking important decisions such as working in collaboration with an international promotor that resulted in a co-tutelle with the University of Lille1 in France. A huge thanks to Stéphane Ducasse, my promotor at the RMoD Inria Team, who came up with uncountable ideas for tackling the problem I worked during my research. I cannot express my gratitude for all the time and effort he put into supervising my work, for the many long and motivating meetings and for giving me a space in his office with a large screen to work each time I went to Lille.

A special thanks to Andy Kellens, my co-promotor, colleague and friend who was always there for me, even when I was mad at him because he gave me more and more work. He survived sharing an office with me for 4 years and I know I did not make it easy for him, but luckily he has a lot of patience. I really appreciated his invaluable feedback during the whole writing process. Even until the last day when he was in Chile working on something else, he spent a lot of time reading my last changes. Thank you Andy!!

I thank Jurgen Vinju, Michele Lanza, Nicolas Anquetil, Julia Lawall, Viviane Jonckers and Ann Nowé for being part of my jury, for taking time to read my thesis and for giving insightful comments during the defense.

Thanks to the CAMP group and other colleagues that helped me to prepare for my private defense. You definitely contributed to my success that day. I would also like to thank my colleagues and ex-colleagues at SOFT and RMoD for their support during these years and for making my integration in the Belgian and French cultures easier.

Being far away from your loved ones is not easy. But their unconditional support at a the distance makes everything possible. I really thank my family for that. Gracias papi Eduardo y mami Emma por su amor, dedicación y por enseñarme desde muy pequeña que la educación es invaluable. A ustedes les dedico este logro y espero que esten orgullosos de su hija. Gracias a mis hermanos, Lore, Maga, Edu y David por compartir mis sueños y festejar mis logros. Gracias a mi tía Olga que siempre esta pendiente de mi. Gracias a todos mis tíos, tías, primos, primas y a todos mis amigos que me mandan su cariño a la distancia. Gracias a mis amigos latinos, especialmente a los ecuatorianos que he conocido en Bélgica y hacen que no me olvide de mi tierra.

Last but certainly not least, I thank Bart, for his support and love during all this time. He has witnessed how the life of a PhD student can be and has helped me in many ways. He has patiently heard me talking uncountable times about my research, he has proof-read several of my papers, he has cheered me up when I was down, and so on. Bedankt Bartje voor alles ;) I also want to thank his parents Marcel and Yolande, and all the Detry family, who have always been eager to celebrate with me whenever we got the opportunity. All of you have made Belgium my second home.

Verónica Uquillas Gómez
Brussels, September 17th 2012

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

Contents

*Modern software is built by teams of developers that work in a collaborative environment. The goal of this kind of development is that developers can alter a set of shared artifacts and inspect and integrate the changes of other developers. At a technical level, such a collaborative development process is supported by version control systems. Since these version control systems allow developers to work in their own branch, merging and integration have become an integral part of the development process. While this development process offers numerous advantages, the activity of merging and integrating changes is hampered by the lack of comprehensive support to assist developers in these activities. For example, the integration of changes can have an unexpected impact on the design or behavior of the system, leading to the introduction of subtle bugs. Furthermore, developers are not supported when integrating changes across branches (cherry picking), when dealing with branches that have diverged, or when assessing the impact and dependencies of changes. In this dissertation we present an approach that aims at alleviating these problems by providing developers and, more precisely, integrators with semi-automated support for assisted integration. We focus on helping integrators with their information needs by means of characterizations of changes and streams of changes (*i.e., *sequence of successive changes within a branch) together with their dependencies. These characterizations rely on the representation of systems' histories and changes based on program entities and their relationships rather than on files and text. Furthermore, our approach also provides tool support that enables integrators to analyze the characterizations and changes.*

1.1 Research Context

Nowadays, software systems play a fundamental role in our technologically evolved society. It runs our cars, TV, and phones, we use it at work, to plan travel, to buy groceries and to keep in touch with family and friends. The development of software systems is no longer limited to single developers working in isolation, but currently the development process takes place in a collaborative context with the participation of teams of developers working together.

Because of the importance of software systems, they are expected to be effective, error-free, adaptable, maintainable, and so on. This is required throughout the lifecycle of the system. Consequently, it is inevitable that a software system needs to be evolved in order to cope with changes in its environment. For example, the requirements of the system might change over time, faults need to be corrected, the system might be expected to run on different hardware or interoperate with other systems. All such changes need to be successfully executed to prevent the system from becoming obsolete and unusable. At the same time, the complexity and size of systems increase resulting in a major effort for the developers in understanding the source code and the evolution of a system.

To cope with the evolution of systems in a collaborative development process, teams of developers use version control systems (such as CVS[1], Subversion[2], Git[3], Mercurial[4], and so on) to work together on a shared or distributed code base. These systems have become an indispensable tool for enabling them to work in a collaborative context. They provide developers with a means to work independently of each other on the same or different artifacts of the system with the intention of later integrating their changes to the source code.

Hence, the integration of changes is a key activity within the software development process. The use of version control systems enables branching and merging. Developers are allowed to work in separate *branches* of the system that later can be *merged* into the mainline of the system. That means, developers publish their code into a repository, and integrators validate and merge such code into the mainline.

Version control systems provide developers with facilities for managing the source code of a system and maintaining that system's history within versions. They also offer features to explore the changes between versions, provide conflict analysis, and elementary merging support. Unfortunately, most of the version control systems only support these tasks at a textual level. That means the program entities and their relationships of the system are not taken into account, therefore they do not consider the semantics of such system. This makes branching and integration a complex problem, since integrators lack support to perform integration activities which is aggravated when integrating across branches.

1.2 Supporting Merge Challenges

Integrators deal with the integration and merge of source code changes that represent bug fixes, enhancements, adaptations or new features into the mainline of a system. Automatic and advanced merging algorithms help developers to merge their modifications in the development repositories. However, there is no adequate support to help integrators take decisions about the integration of published merged changes into the mainline.

The current state-of-the-art consists mostly of tools that do not provide an overview of the changes (how changes are distributed?, what groups of entities did change?, what other changes are depending on a particular change?). At the same time, existing tools do not offer the possibility to understand changes within their specific context. Almost invariably, integrators need to manually read the

[1]CVS: http://savannah.nongnu.org/projects/cvs
[2]Subversion: http://subversion.apache.org
[3]Git: http://git-scm.com
[4]Mercurial: http://mercurial.selenic.com

changed code, check the diffs, and dig up details from unchanged code in order to build an idea of a change and to understand the context of such change.

This situation can become more complicated when developing large systems. The task of merging remains mostly manual and tedious due to a lack of practically applicable advanced tools. First, merging techniques used by popular version control systems are based on simple, text-based algorithms, and are therefore oblivious to program entities and the relationships that are merged. Even though there exist other approaches providing advanced merging support [Apel 2011, Mens 2002] that significantly reduce the amount of merging conflicts, such approaches do not support integrators in identifying redundant changes or changes that introduce inconsistencies at the level of the design of the target system. Second, there are no analyses to identify and understand the dependencies between changes. For example, there is no support to determine the dependencies of a change within a branch that are needed in order to merge that change with another branch. As a change may require prior changes, the integration can produce a functional system if such prior changes are merged as well. Note however, that this not necessarily means that the system will be 100% correct. To determine dependencies the integrators are left to manually compare changes within the input stream of changes, and assess how these changes may impact the target system. Such work is particularly tedious between product forks, where the distance between branches grows larger over time.

In addition, there is little support out of the box to be able to perform queries and analyses over the complete history of a system to support developers and integrators in understanding changes and therefore the evolution of a system. To support such analyses and more precisely to support integration of changes, information that is not readily available from version control systems is required.

Developers build tool support with their own infrastructure and history analysis on top of the version control systems [Zimmermann 2004b] to perform particular analyses, for example, finding program entities that co-evolved together. However, certain analyses are not straightforward, like comparing all the differences between all the senders of a given method in past versions, or providing analysis to support cross-forks merging – which is an even more complex scenario as the branches drifted apart. To facilitate history and change analyses we need adequate source code models to represent program entities and their evolution. Moreover, we have to take into account which definitions, abstractions and APIs for such models are needed.

Software developers face the complex task of understanding changes in evolving systems to be able to keep them usable. The integration of changes is part of the evolution of a system, and hence support should be provided to integrators to assist the integration process.

1.3 Pharo Smalltalk as a Testbed

As mentioned before, integration is a key task in the development process of any software project and lacks adequate support for developers that deal with integration activities. This problem exists independently of the programming language or development platform used by developers and integrators. Moreover, this problem is aggravated in a collaborative development environment that relies on branching and merging to allow teams of developers to work together.

Our research aims at assisting integration of changes in a realistic context by means of scientific methodologies. Hence, for this dissertation we have chosen a concrete case of the problems aforementioned, the Pharo project (http://www.pharo-project.org). It is an open-source Smalltalk platform that not only illustrates the integration problems but also serve us to evaluate our solution with its community. This provides us with real feedback from developers that actually face the inherent problems of branching and merging.

In the following, we provide an overview of our motivation for choosing Pharo [Black 2009] as a case study, and discuss how this impacts the generalizability of our work.

Motivation for using Pharo as a case study

- Pharo is a big open-source project that resulted from branching Squeak in 2008. Since then both projects have evolved in isolation of each other. In Pharo branching is extensively used to incorporate the contributions of developers for the different versions and to merge changes that happened in Squeak that may contribute to existing shared features. Pharo lacks support for cross-branch integration, performing cherry picking from several fork systems such as Squeak or Cuis is a difficult task as these systems have diverged considerably from Pharo.

- Pharo provides us with access to developers and mainly to the integrators of the core development of several projects. As we are active members of this community, we have easy access to the community members for evaluating our work. What is also important for the evaluation of our work is that the core developers are eager to try approaches that assist them in their development activities. Furthermore, part of our infrastructure is already integrated with the core of the project. This illustrates that potentially our approach has considerable impact on this community.

- Pharo provides its own development environment and tools can be easily integrated with it.

Furthermore, we have integrated our approach with the underlying object-oriented programming language of Pharo, namely Smalltalk, due to technical reasons. Smalltalk is a reflective and small language (Smalltalk defines 12 types of AST nodes vs the 83 types of AST nodes in Java[5]), therefore it is adapted for research prototypes. Still Smalltalk is a full object-oriented language, simple but not simplistic.

Pharo also gives us access to other platforms such as Moose [Nierstrasz 2005], a data and software platform (http://www.moosetechnologgy.org). It provides a rich set of analysis tools and frameworks. To name two that we use for building our tool support: Glamour, a powerful browser builder engine [Bunge 2009] and Mondrian, a scripting-based visualization engine [Meyer 2006,Lienhard 2007].

Generalizability of our approach

Due to the increased popularity of distributed versioning systems such as Git, teams of developers are more often developing in their own branch. Therefore, the whole integration process used by Pharo is very similar to what is happening in other development communities. In that regard, we believe that similar integration problems occur outside the context of Pharo. Moreover, considering that Pharo is a complex ecosystem consisting of different projects, where each project has the same kinds of integration issues, we believe that our contribution goes beyond the Pharo community.

[5]Java DOM/AST: http://help.eclipse.org/helios/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/package-summary.html

The conceptual approach proposed in this dissertation, as detailed in the next section, is agnostic to the programming language and technology used. The whole idea of characterizing changes and streams of changes can therefore also be applied to other languages such as Java, C# etc. Note however that our technical realization of this approach is implemented within Pharo and is as such tightly woven with this environment and underlying programming language.

1.4 Approach

We present a visual overview of our approach in Figure 1.1. This map shows the interaction between the different contributions of this dissertation.

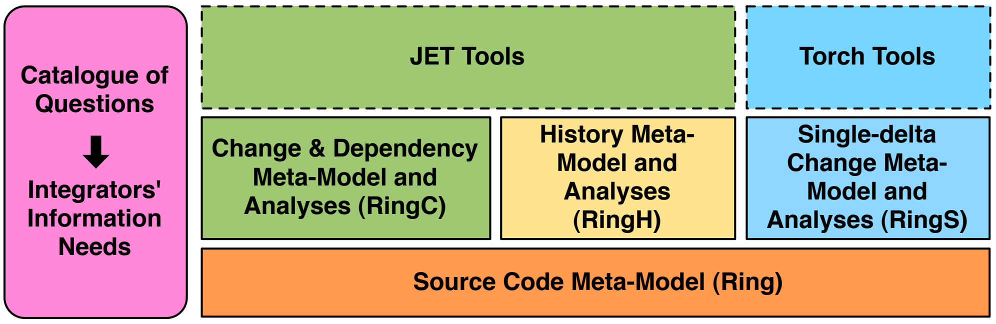

Catalogue of Questions
Integrators' Information Needs
JET Tools
Torch Tools
Change & Dependency Meta-Model and Analyses (RingC)
History Meta-Model and Analyses (RingH)
Single-delta Change Meta-Model and Analyses (RingS)
Source Code Meta-Model (Ring)

Figure 1.1: *Contributions map.*

This dissertation starts with an analysis of the integrators' information needs. To achieve this we performed a survey among professional developers that integrate changes as a means to gather the questions they raise during the different activities involving the integration process (*e.g.,* understanding, assessing or cherry picking changes). Based on the catalogue of questions we determined the kinds of information required for supporting answering these questions. We obtain these kinds of information by providing a representation of the history and changes of a system, along with the dependencies between changes from source code histories stored by version control systems.

To support our ideas and achieve assisted integration we built an infrastructure consisting of several meta-models to serve as the underlying models for our analyses, and tool support as clients of such models. Precisely, our approach consists of a number of components: (a) the source code, history, change and dependency meta-models, (b) the version comparisons and dependency analyses, and (c) the tools for characterizing a single delta (*i.e.,* changes between two versions), streams of changes and the dependencies between changes within the stream. These tools were defined to semi-automatically assist integrators and as a way to validate our approach and results. Currently, our approach allows integrators to analyze changes and make integration decisions separately from the version control system.

In what follows, we take a more detailed look at each of the different components of our approach.

Catalogue of integrator questions and integrators's information needs. We defined a catalogue of 64 questions from a survey applied to members of three Smalltalk communities as a means to gather questions that developers often raise when integrating changes. This catalogue serves us to identify the integrators' information needs required to answer these questions and assist the integration process within a branch and across branches. We use such information to provide a characterization of changes between two versions and characterization of changes within a stream of changes.

Ring. To give integrators access to their information needs, we require to provide a meta-model. We aim at providing a meta-model consisting of several layers, the first of which being *Ring* as shown at the bottom in Figure 1.1. We defined *Ring*, a simple unified source code meta-model for representing object-oriented software systems. It serves as the underlying meta-model of the history and change meta-models and as the basis for several analyses and tools. In addition, *Ring* was developed to be used by other Pharo tools, for example to support remote browsing.

Ring was defined after performing an analysis of several source code meta-models used mainly by core tools in the Pharo Smalltalk environment (*e.g.,* the versioning system, the refactoring engine) and offered by external frameworks in the Smalltalk community (*e.g.,* Moose). *Ring* was later integrated as the source code meta-model of choice for tool interaction in Pharo 1.4.

RingH. To represent the history of software systems and provide analyses of such histories, we defined the *RingH* history meta-model on top of *Ring* (shown in yellow in the figure). *RingH* models source code entities such as packages, classes, methods or attributes as well as the relationships between such entities such as class inheritances, method calls, class references and attribute accesses etc. We extract system's history models from the source code history contained within versioning repositories.

RingS and Torch. To assist integrators in comprehending source code changes between a pair of versions (*i.e.,* delta) and therefore helping them in answering questions related to single deltas and in taking decisions about the integration within a branch, we defined *RingS* and *Torch* (shown in light blue in the figure). *RingS* is a change meta-model that defines the changes and their context within a single delta. It is built on top of *Ring* and serves as the underlying meta-model for version comparison analysis.

Torch provides tool support to characterize single deltas based on a *RingS* model. It uses several of the integrators' information needs to provide such characterization of changes and by means of visualizations aid integrators to comprehend these changes and their context. *Torch* provides a visual dashboard that not only shows the changes between two versions but also the version in which those changes were applied.

The *Torch* dashboard presents metrics about changes, a set of visualizations showing the structural information of changes, symbolic clouds of the changed source code, several panels to explore the details of changes, and two diffs as a fly-by-help to quickly explore information of changes. By combining graphical and textual information, *Torch* brings semantic information to change exploration.

For the evaluation of *Torch*, we presented several usage scenarios, and we performed a field evaluation (questionnaire) with 6 integrators of the Pharo community, and pretest/posttest pre-experimental study with 10 developers (including non-Smalltalk developers).

RingC. To assist cross-branch integration our approach defines a representation, analyses and tool support that provides integrators with their information needs regarding stream of changes that may be integrated across branches. For this we define *RingC* and *JET*, both shown in green in the figure.

RingC is a change and dependency meta-model that defines the changes of a sequence of successive versions as they occur in a typical version control system. *RingC* is built on top of *Ring* and utilizes the information contained within the *RingH* model. It serves as the underlying meta-model for delta and dependency analyses.

JET. *JET* is dedicated to characterizing deltas, dependencies between changes, and dependencies between deltas within a stream of changes. It is built on top of *RingC* and aims at assisting integrators in answering questions regarding stream of changes and cherry picking changes.

JET provides a set of tools to support integrators in understanding stream of changes and their dependencies. The dashboard offers exploration of deltas, their changes and their dependencies together with metrics about changes. The dependency map gives a visual overview of the dependencies between deltas. The query browser explores the whole evolution of a particular change within the stream, and few utilities to compare any change with the current system in which it would be potentially integrated. Moreover, *JET* is integrated with *Torch*.

For the evaluation of *JET*, we performed a qualitative evaluation with a Pharo integrator and developer on a five-year stream of changes from the Squeak project with the goal of integrating it in Pharo.

1.5 Contributions

We present our contributions classified in two categories: conceptual and technical contributions.

Conceptual Contributions

- An in-depth analysis of the problem underlying this research.
- A catalogue of 64 integrator questions by a questionnaire filled out by three Smalltalk communities.
- Based on our catalogue, a characterization of the integrators' information needs.
- A first-class representation of the history and changes of a system in order to obtain the information required to provide integrators' information needs.
- We support assisted integration by providing a characterization of changes and, by means of a dependency analysis between such changes for characterizing streams of changes.
- Use of visualizations to understand changes and observe patterns in the changes.
- A qualitative evaluation of our approach and tools.

Technical Contributions

We have developed a tool suite to support our approach. It is tightly integrated with the Pharo development environment. This tool suite consists of a concrete implementation of the *Ring* source code meta-model (which is also part of Pharo), three models on top of it (the *RingH* history meta-model, and the *RingS* and *RingC* change meta-models), and two tools (*Torch* and *JET*) to aid in characterizing changes and streams of changes. In addition, both tools were also used in our evaluation.

1.6 Structure of the Dissertation

This dissertation is structured as follows:

Chapter 2. We start this dissertation by presenting an analysis of the inherent problems related to integration. Concretely, we discuss collaborative software development and how version control systems provide means to support integration of changes in a collaborative environment by using branching and merging. We introduce a concrete example to motivate this dissertation, the integration problems, the challenges to support integration and the requirements of our solution.

Chapter 3. In this chapter we introduce the definitions and terminology used in our dissertation. We describe our study that resulted in a catalogue of integrators' questions that are raised when performing integration activities. Based on these questions we identify and describe the integrators' information needs that can be used to support answering such questions, and therefore to characterize changes that are the requirements of our solution. In this chapter we also provide a literature study of related work by surveying other approaches that support history and change modeling, merging, change impact analysis, change dependencies and understanding developments tasks.

Chapter 4. This chapter is dedicated to describing source code modeling and *Ring*, our source code meta-model. *Ring* is a technical contribution of this dissertation defined to serve as the foundation for other contributions such as meta-models and analyses. We start by describing data models used by several version control systems and source code meta-models proposed by different approaches that inspired *Ring*. We describe the architecture of the *Ring* source code meta-model and illustrate it with two concrete examples of simple tools built on top of *Ring*.

Chapter 5. After having introduced our source code meta-model, we define *RingS*, our change meta-model for representing the differences between pairs of versions. *RingS* is built on top of *Ring* and it is the underlying meta-model to enable version comparison analysis. We describe a concrete client of our change model, namely *Torch*. This is our research prototype that is implemented in Pharo Smalltalk to provide characterization of changes within a single delta and tool support. The use of *Torch* is illustrated in several examples that show real integration scenarios. We present a field evaluation and a pre-experimental user study performed to assess *Torch*. We conclude this chapter by providing a comparison of our approach for characterizing changes with several approaches we discussed in Chapter 3.

Chapter 6. Going further to represent histories and streams of changes of a system, in this chapter we describe our history meta-model, namely *RingH*, our change meta-model, namely *RingC*, and the history and change dependency analyses. *RingH* and *RingC* are built on top of our *Ring* source code meta-model. We present the architecture of *RingH*, how we import the history of a system, and how objects are created and queried within a history model. Next, we present the architecture of *RingC*. This change model differs from *RingS* in the sense that it is oriented to represent a stream of changes (*i.e.,* sequence of deltas) and it is derived from the history of a system, rather than from pairs of versions. Finally, we describe our analyses for calculating deltas and dependencies within a stream of changes.

Chapter 7. In this chapter we describe *JET*, our approach for characterizing streams of changes and tool support. *JET* is a tool built on top of the *RingC* change model-model and an application of the *RingH* history meta-model. It is our research prototype that is implemented in Pharo Smalltalk to provide the integrators' informations needs and assist cross-branch integration (*e.g.,* cherry picking). We describe how *JET* characterizes deltas, dependencies between changes, and dependencies between deltas. The *JET* tools – the dashboard, the map, and the query browser – are explained. We discuss how *JET* supports answering questions related to stream of changes. This is followed by the qualitative evaluation of *JET* based on a five-year stream of changes and performed by an integrator and a developer. We conclude this chapter with a comparison of our approach with several approaches we discussed in Chapter 3.

Chapter 8. We present the conclusions of this dissertation and revisit the catalogue of questions as a means to show how our approach can assist integration. We discuss some of the limitations of our approach and implementation and suggest how we can overcome these limitations. We also propose a number of directions of future research. Finally, we summarize the conceptual and technical contributions of this dissertation.

Use of the Contributions Map. We show the *contributions map* (Figure 1.1) at the beginning of each chapter that describes our approach. We show in light grey the contributions that were already explained, in their original color the contributions that will be explained in that chapter, and in dark grey the contribution that are still pending of explanation.

CHAPTER 2
Problem Analysis

Contents

Overview

This chapter introduces the problems that developers face when integrating changes. The goals of this chapter are threefold. First, we present how current software development is driven by a collaborative environment which allows multiple developers to share artifacts and work without interfering with their peers' work. Second, we explain how this collaborative development is supported by the use of branching and merging and what are the inherent problems of branching and merging. By means of a concrete example we illustrate the use of branching and motivate the need for supporting integration. Third, we introduce the challenges that have to be tackled to assist integration and the requirements for a solution that can (semi-)automatically support developers integrating changes. Such requirements are the motivation for the next chapters of this dissertation.

2.1 Collaborative Software Development

Depending on the complexity or the size of an application, the development and maintenance of such applications may require not one developer but teams of developers working in parallel. Parallel development has become a common phenomenon in the development of large-scale software systems [Thione 2005]. Modern software development is highly cooperative, with co-workers potentially being located on different continents in different time zones.

Large-scale software development involves teams of developers working on shared artifacts from a single code base. In this environment, a principled approach is necessary (*e.g.,* the divide and conquer approach). A developer may work on a feature of the system while another developer may be working on a different feature of the same system. They are working in parallel and once they are done, their changes need to be merged. To support their development tasks they usually rely on an infrastructure, such as version control systems (*e.g.,* Subversion) which allow them to share and integrate artifacts.

Developers follow a software development model. For example, the *collaborative software development model* which is a style of software development whose focus is on public availability and communication, usually via the Internet. Here multiple developers around the world can contribute to a system. This model is mainly used for freeware, open-source software, and commons-based peer production.

How the development model is structured and supported has a profound effect on both the quality and timeliness of the product. In general, team development reduces the time to market, but the cost of coordination problems introduced by duplicate and conflicting changes in the shared artifacts can be nontrivial [Perry 2001]. Even though each developer may have a specific task, developers may be affected when their changes conflict with the changes introduced by others.

2.1.1 Version Control Systems

Version control systems (also known as *Revision Control Systems*) allow users to track and store changes, collaborate and share project files. These files are mostly source code files, but they can also be any document or other collections of information related to a project. Every change made to the files is tracked and identified by a key, along with who made the change (*i.e.,* author), when he made it (*i.e.,* timestamp) and why he made it (*i.e.,* message specifying the reason of the change: problems fixed, enhancements introduced, added features, adaptations). When the changes are stored, they are known as revisions[1]. Revisions can be compared, restored, and merged. However, these actions over files are done at a textual level.

Version control systems are essential tools for any form of distributed, collaborative development where several teams may change the same files at the same time (*e.g.,* a configuration file). These systems do apply for any software development project, no matter whether it is a simple web page or a large application. They keep data that represent the development life cycle of a system and so, how that system evolved. They provide a rich resource that can be used to assist developers in their implementation and maintenance tasks.

Source Code Control System (SCCS) [Rochkind 1975] was an early revision control system that was predominantly used in the Unix community until the release of the Revision Control System (RCS) [Tichy 1982]. Even though RCS only operates on single files, it is still used as part of the GNU project. In 1986, the Concurrent Versions System (CVS) was released to deal with multiple files. It was designed as a centralized system for sharing information. Subversion is a successor of CVS and is probably the version control system with the widest adoption. Many other version control systems have been created specially to support a distributed approach, such as Git[2], Darcs[3], Mercurial[4], Bazaar[5], which are very popular in the open-source community.

2.1.2 Definition of Branching and Merging

Branching. In version control, branching is the process of creating a *copy* (branch) of a code base (main branch) for allowing development teams or individual developers to work on a branch in parallel

[1]A *revision* is the source code in a version control system at a given point in time.

[2]Git: http://git-scm.com

[3]Darcs: http://darcs.net

[4]Mercurial: http://mercurial.selenic.com

[5]Bazaar: http://bazaar.canonical.com

and in isolation to the main branch. Multiple branches can be created out of the main branch to allow several developers to work independently of each one, or to maintain the different releases of a software product family.

Branching implies the ability to later merge changes back into the main branch or any other branch. Branches that are not intended to be merged are usually called *forks*. Such branches often become independent systems that serve different purposes, such as supporting customizations, testing out new technologies, and so on.

Merging. In version control, merging is the key process that reconciles multiple changes applied to a collection of files from multiple source branches into a single target branch. For example, if a file is modified by two developers in two different branches both versions of the file need to be merged. When two branches are merged, the result is a single collection of files that contains a subset of both set of changes.

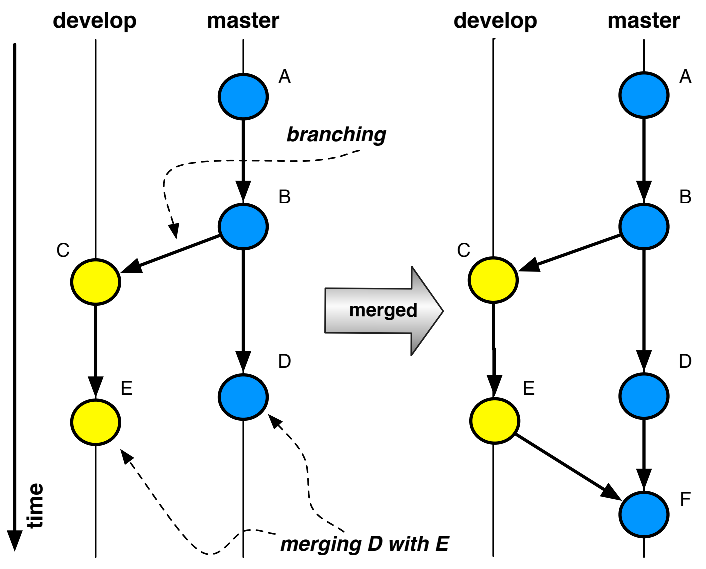


Figure 2.1: *Branching and merging.*

We illustrate *branching* and *merging* in Figure 2.1. Here we have two branches (*master* and *develop*) and their versions shown in blue and yellow. Note that the *develop* branch was created from the *master* branch at version *B*. Later on, both branches evolved to versions *E* and *D* in the *develop* and *master* branches respectively. Finally, the version *D* in the *master* branch is merged with the version *E* in the *develop* branch resulting in the merged version *F* in the *master* branch.

Conflicts can appear when merging changes that are incompatible (*i.e.,* changes that overlap). For example, in one branch the line #10 was removed from file *f*, while in the other branch the same line was modified in file *f*. In this case, a conflict is raised and the merge cannot be performed automatically, and someone must decide manually exactly what the resulting file *f* should contain.

2.1.3 Version Control Systems Support Branching and Merging

Version control systems such as Subversion, Git, etc. support branching and merging. What's more, branching is intrinsic to distributed version control systems where there is no a central repository at a technical level. Instead, each developer maintains his own repository from which other developers may merge changes. Such systems are typically used as part of a collaborative development model, in which hundreds or even thousands of people are working towards the release of a single product. Git [Git 2005], which is becoming increasingly popular, has placed branching at the center of its architecture and philosophy. GitHub[6] gives an idea of the importance of this practice of branching and merging, with some projects having thousands of branches such as Bootstraps (> 4,000), homebrew (> 3,500), Ruby on Rails ($\simeq$ 2,800), and 9 others with more than 1,000 branches.

Merges are distinguished by their directionality (pulling or pushing changes). *Downstream* merges pull changes into lower branch levels (*e.g.,* from the main branch to a feature branch). Conversely, *upstream* merges push changes into higher branch levels (*e.g.,* from a develop branch into the main branch). The first case is more typical of distributed version control system such as Git, whereas, the second case is more typical of centralized version control system such as Subversion.

Merging changes in a large-scale collaborative software development environment poses substantial challenges [Phillips 2011, Perry 2001]. On the one hand, problems arise when dealing with multiple branches of a software product family, for example when sharing and reusing common code, propagating common changes across different versions, and identifying the version suited for a given application. On the other hand, dealing with temporary branches that contain changes that are meant to be merged raise the problem on figuring out how to merge the multiple version back into a coherent single version, resolving potential conflicts that might arise in the process.

Concretely, if multiple developers concurrently modify the same code (or file) in different ways, the version control system has to determine how to merge them, or must report a conflict to be manually resolved. More subtle challenges arise, however, when disjoint code fragments change but there are dependencies between them that may cause erroneous program behavior, without reporting conflicts. When developers are working towards the common goal of ultimately releasing a single piece of software, they must actively try to maintain compatibility with the mainline; if they diverge too far from the overall software design, or make too many changes before communicating those changes with others, merging such changes may not be possible.

2.1.4 Why is Branching Used?

Branching is used in the open and closed source development worlds. As explained before, it allows teams or individual developers to work in parallel on the same project by each having a copy of the code that can be modified without interfering with others' work. It also allows developers to keep different releases of a software product family. Creating branches is therefore motivated by the need to prevent workflow disruption and reduce overall cycle-time, by allowing developers to stabilize their development in isolation before integrating with the main branch. That means, if the goal was to decompose tasks among developers to contribute to one project, their changes will have to be integrated later on.

However, branching is not limited to the case of multiple developers working together towards a common goal or software product family. Another, arguably more powerful, form of branching occurs when software developers want to benefit from the maturity and advanced functionalities of an existing project to rapidly create a new project and adapt it to other specific needs. These new branches are the ones known as forks.

[6]GitHub (online project hosting site): https://github.com/popular/forked

Many factors can intervene in the creation of branches that are oriented to become independent projects. For example, a specific client's requirements conflict with the project goals, or economic factors interfere, as when an open-source producer is purchased by a company that does not have a history of supporting open-source development.

Branches can be created, maintained and evolved with no intention of ever merging back with the original code base. Because this form of software development starts with a fully functional code base, its impact can be rapid and enormous. For example, Android began in 2003 as a branch of the Linux operating system to take advantage of Linux's reliability, performance, and advanced functionalities. By branching, Android was free to make design decisions that were critical to meeting the constraints of the mobile environment, in contrast to Linux, which targets the entire computing spectrum from embedded systems to supercomputers. At the end of 2011, four years after its first release in 2007, Android had captured 43% of the US mobile market[7], 50% more than its nearest competitor.

Even if the initial goal of creating a branch is to allow developers to pursue a separate agenda from that of the original software project, if the split is amicable, the common origin of the source code means that the new branch can still benefit from bug fixes and new functionalities developed for other branches. The process of selectively applying code changes from one branch into another is known as *cherry picking*. Cherry picking suffers from the same problems of conflicts as those cited before for the distributed collaborative development of a single software project. But these problems are substantially compounded as the branches involved naturally evolve and drift apart. Over time, it becomes increasingly difficult and tedious for a developer to determine whether a change from another branch can benefit his branch, whether the dependencies of that change exist in his branch, whether the change will introduce bugs into his branch, and how the change relates to any modification he has done in his branch.

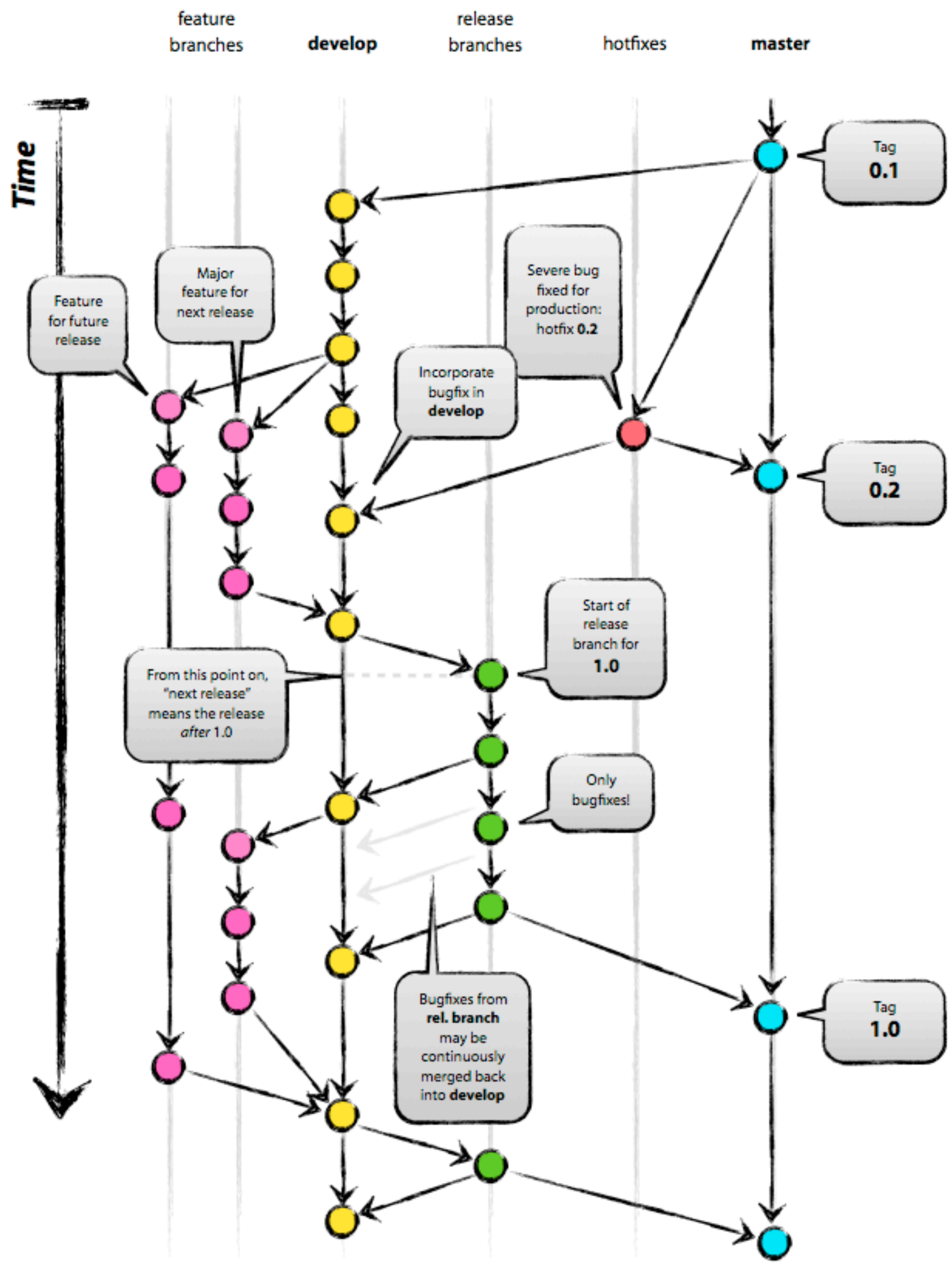
2.2 Supporting Collaborative Development Through Branches

Managing multiple branches in the development process and merging changes from one branch to another can be a sophisticated and complex task [Phillips 2011] as can be gathered from Figure 2.2 [8]. This figure illustrates a proposed software development model using the Git version control system and relying heavily on branching.

Figure 2.2 shows two main branches: the *master* branch (right, blue) that holds the production-ready version of the project; and the *develop* branch (third from left, yellow) holding the next release version. When the source code in the *develop* branch reaches a stable point all the changes should be merged back into *master* and then tagged with a release number. Several features are developed in their own *feature* branches (first two from left, pink). They are ultimately merged back in the *develop* branch, or discarded if unsuccessful. The *release* branch (third from right, green) holds a short lived version of the *develop* branch system when it is almost ready to be released. It allows one to iron out the last bugs and is not intended to receive any new features. One can see it as holding the release candidates. Finally, the *hot fix* branch (next to last, red) as its name indicate allows a developer to fix critical bugs on a production version without disrupting the development of new features.

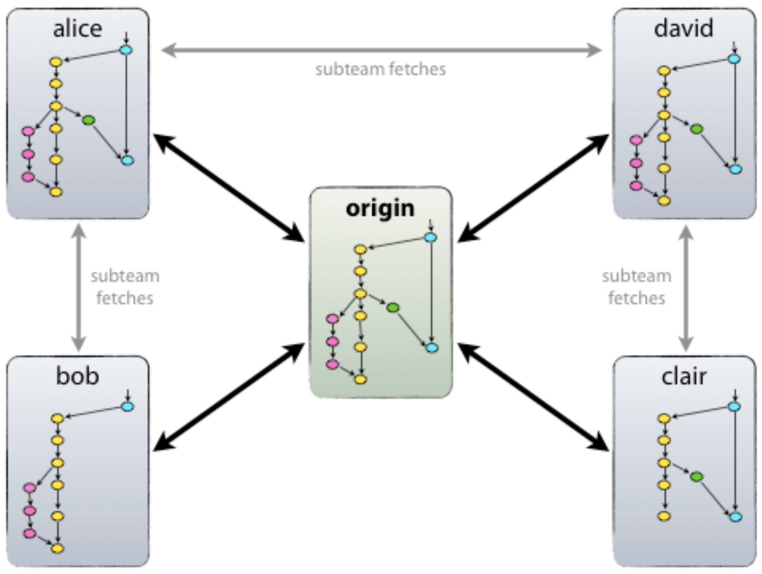
[7] http://www.pcmag.com/article2/0,2817,2395804,00.asp

[8] Taken from http://nvie.com/posts/a-successful-git-branching-model


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Major feature for next release
Feature for future release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Author: Vincent Driessen
Original blog post: http://nvie.com/posts/a-successful-git-branching-model
License: Creative Commons CC By-SA

Figure 2.2: *Development process driven by branches.*



Author: Vincent Driessen
Original blog post: http://nvie.com/posts/a-successful-git-branching-model
License: Creative Commons CC By-SA

Figure 2.3: *Distributed development processes.*

Figure 2.3 shows how a team of developers may work in a distributed environment. In this case, four developers (Alice, Bob, David and Clair) pull and push changes to a *central* repository (not at a technical level) which is tagged as *origin* in the figure. It also shows that developers may form sub-teams (Alice and Bob, Alice and David, and Clair and David) by pulling changes from peers. Sub-teams allow small group of developers to work together on particular features before pushing their changes into *origin*.

Defining an adequate process and branching strategy to master the complexity of parallel development of bug fixes and new features is a common problem, other solutions, similar to the one illustrated here, may be found, *e.g.,* [Walrad 2002], [AmcomTechnology 2010]. For lack of more advanced solutions, these approaches propose to use the branching mechanism of the configuration management tool to help identify individual features and/or bug fixes.

From a purely textual point of view, one may retrieve the changes between any two revisions of a software system in a branch and try to apply it to a revision in another branch. This is what a developer hopes to achieve, for example in Figure 2.2, when merging a *feature* branch into the *develop* branch. In practice, this works well if the two branches are not too different from one another and if we only consider textual representation. As soon as we consider the semantics of the resulting program, a simple modification from protected to private in Java can lead to unexpected failures such as the *fragile base class problem*[9] [Steyaert 1996]. In the case of forks, the complexity is much greater because we consider two *master* (or *develop*) branches, each having its own release and feature agenda. Solutions relying on a specific software process are not applicable here because the two branches are managed by two independent communities. As a consequence, the developers require more information to understand what are the prerequisites (*e.g.,* dependencies) to integrate a

[9]The *fragile base class problem* is a fundamental architectural problem of object-oriented programming systems where base classes (superclasses) are considered *fragile* because a seemingly safe modification to a base class, when inherited by the derived classes, may cause the derived classes to malfunction.

change developed for another *master* branch and what impact this change may have on their code.

To illustrate the use of branching, we analyze a concrete example in the next section.

2.2.1 Concrete Example: Pharo

Pharo[10] is an open-source distribution of a new generation Smalltalk system. It is a platform composed of a set of core libraries (compiler, stream, scheduler, collections, events, etc.), a large UI framework, and a large set of tools such as code browsers debuggers. Pharo was born in 2008 as a fork of Squeak[11], the Smalltalk system originally developed by the team of Alan Kay. Other branches of Squeak are also under development including Croquet, Cuis, Etoys, as shown in Figure 2.4. Around 1000 packages, for various purposes, have been developed on top of Pharo. Often such packages can also be used with the other Squeak branches as well as on GemStone[12] (a proprietary Smalltalk system).

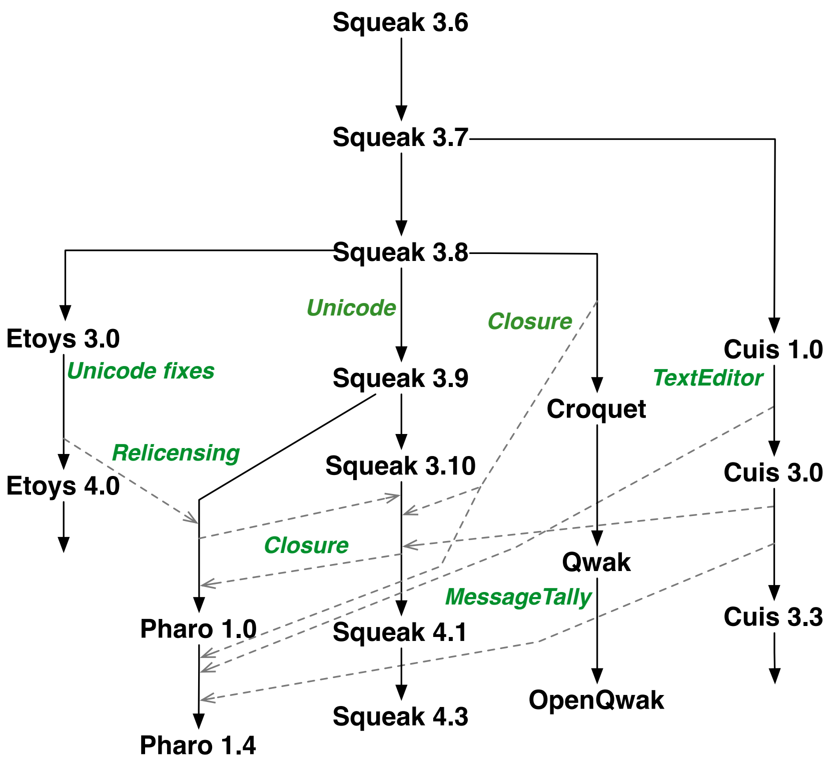


Figure 2.4: *Pharo / Squeak fork ecosystem: cross branching.*

Each of the Squeak branches is continuously evolving. The Pharo community actively integrates changes from Cuis and Squeak. The Squeak community occasionally integrates changes from Pharo.

Pharo itself has evolved rapidly in 4 years and its community has grown considerably as well. People from academia (12 research groups) and industry (34 companies) are contributing with bug fixes, enhancements and new features. Pharo has 173 official contributors[13]. Currently, the Pharo

[10]Pharo: http://www.pharo-project.org
[11]Squeak: http://www.squeak.org
[12]GemStone: http://www.gemstone.com
[13]Pharo records obtained on May 31st, 2012

Consortium is being established among users and industrial partners to sustain the development of Pharo.

Branching in Pharo

The evolution of Pharo is shown in Figure 2.5. Here each version is shown with a vertical line divided in two parts: the arrowed dashed line represents the period of time in which its development started until its release (pre-release), and the arrowed line represents the period of time after it was made public until its maintenance ended (post-release).

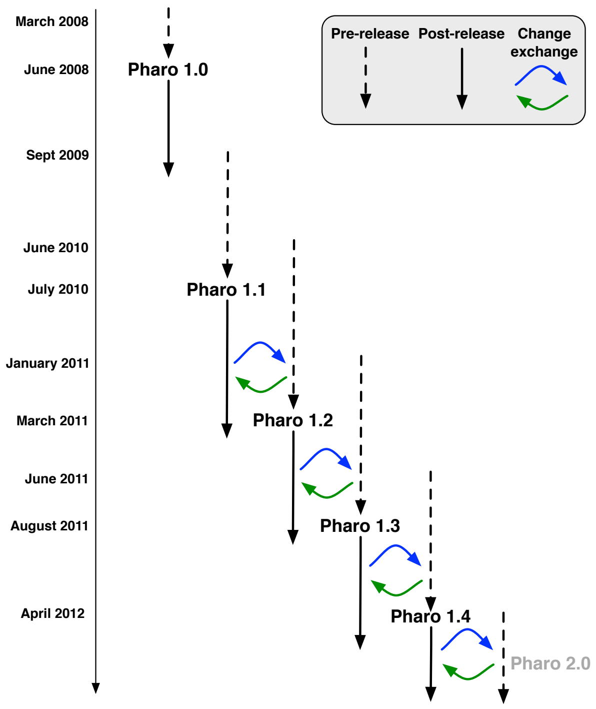


Figure 2.5: *Pharo evolution: internal branching.*

In 4 years the Pharo community has made public five official releases: version *1.0* in June 2008, version *1.1* in July 2010, version *1.2* in March 2011, version *1.3* in August 2011, and version *1.4* in April 2012. Currently, version *2.0* is already under development (shown in grey).

Each version is managed as a separate branch. Note in Figure 2.5, that once a version is released (*e.g.,* 1.3), the main developers of Pharo also maintain the previous version (*e.g.,* 1.2), and continue working on the next version (*e.g.,* 1.4) which in some cases started before the previous one was released. In this situation, the main developers may deal with three main branches at the same time. Moreover, the development of these versions does not happen in isolation. A continuous exchange of changes is done between these branches, as shown with the green and read arrowed lines. For example, between versions *1.4* and *2.0*, the blue arrowed line indicates that changes done in the post-released 1.4 version are integrated into the development 2.0 version, and the green arrowed line indicates that changes done in the development 2.0 version are integrated into the post-released 1.4 version.

The Pharo ecosystem shown in Figure 2.4 and the Pharo evolution shown in Figure 2.5 illustrate the two possible scenarios of integration of changes in Pharo: a) between forks that we call *cross branching integration*, and b) between branches that we call *internal branching integration*.

- *Cross branching integration* is the more complex case, as the changes come from a system (*i.e.,* Squeak) that has separately evolved from the fork in which they may be integrated (*i.e.,* Pharo). Bringing changes from Squeak into Pharo is a very difficult task. In order to integrate such changes, the developers not only need to understand the changes, their context and their evolution in Squeak, but also how those changes can be integrated into Pharo, which other changes are required, and more importantly how those changes will impact Pharo and its ecosystem.

- *Internal branching integration* is done daily and even though changes may come from sibling branches (*e.g.,* from version *1.4* to version *2.0*), the integration of changes is not necessarily easier than in *cross branching integration* as changes at design level are taking place. Still, the developers that are integrating changes from one branch into another need to deal with all the activities involved in the integration process, such as, understanding the changes, selecting the adequate changes, solving merging conflicts between changes, and analyzing the impact of changes.

2.2.2 Integration Problems: Overview

Until now we have presented how branching supports the software development process, in particular in a distributed environment where multiple developers are allowed to work independently of each other in a separate branch. Furthermore, the concrete case of Pharo not only shows how multiple branches are managed for its development but also shows how this process is still impacted by changes applied in other forks such as Squeak or Cuis.

Despite the advantages of branching, it may pose considerable problems before, during and after the actual integration of the work (*i.e.,* changes) done by multiple developers happens. We present an overview of the problems of integrating changes at two levels:

Integrating changes within a branch. We consider this case when multiple developers are working on branches derived from the same main branch, and they are working towards releasing the same product. For example, some features of the Pharo 2.0 version are being developed in branches that in the future will be integrated into the main branch. Here, branches are not evolving independently of each other. This case, however, does not imply that the integration of changes is free from conflicts as several developer may be working on the same feature.

Integrating changes brings other activities such as understanding changes before the integration, merging changes, solving conflicts during the actual merge, testing, and dealing with the impact of changes after the integration. Version control systems do not offer adequate support to cover all these activities, and developers have to perform some of them mostly manually.

Integrating changes between branches: cherry picking. When integrating changes between forks or branches that are drifting apart in their design or development, another problem is present when performing *cherry picking*. Cherry picking describes the action of selecting which changes should be ported from one branch to another.

Cherry picking is difficult and time consuming, considering that a developer has to manually determine whether a change from another branch is relevant, whether the resources the change requires (dependencies) are available in the developer's branch, whether the change will break the invariants of his branch, and how the change relates to any customizations he may have introduced.

Moreover, integrating these changes also suffers from the problems mentioned before. But these problems are substantially compounded as the branches involved naturally evolve and drift apart. For example, merging changes from Squeak done 2 years ago to the latest Pharo version can be loaded with conflicts.

In the next section, we introduce the challenges to support the integration of changes and the problems raised by this process.

2.3 Challenges to Support Integration

Integrating changes that represent fix bugs, enhancements, new features or adaptations due to changing environments are key software development activities. However, integrating these changes in a large-scale collaborative software development environment poses substantial challenges. For example, if two developers modify the same code in different ways, the revision control system has to determine how to merge them, or report a conflict to be manually resolved. More subtle challenges arise, however, when disjoint code fragments change but there are dependencies between them. While revision control systems can help to textually detect conflicts [Berliner 1990], they do not help to identify semantic consequences of a change.

There are several merging techniques: text-based [Leblang 1984, Tichy 1985, Berliner 1990], syntactic-based [Asklund 1994, Buffenbarger 1995], semantic-based [Westfechtel 1991, Binkley 1995], operation-based [Shen 2004, Dig 2008] and merging algorithms such as two-way merge [Hunt 1976] and three-way merge [Lindhom 2001]. The current state-of-the-art however is mostly constituted by textual diff tools, the widely used version control systems use text-based merging techniques where semantics is not taken into account when merging.

Integrating changes is a difficult task since integrating a change requires not only the merging of source code but also *an understanding of the changes and their context, and its potential impact* on the system. This can be more complex than doing the actual merge and there is no adequate support for these activities.

Based on the current state-of-the-art and on the development process we identify two groups of challenges to provide a semantic-oriented support for integration of changes: (a) Change characterization, and (b) Merging support.

In the following, we describe each of these groups of challenges, and we present particular problems in each case together with examples. They illustrate the problems that developers have to deal with when integrating changes.

2.3.1 Change Characterization Challenges

During daily software development activities, it is required to understand changes in order to assess their impact, review their quality, and so on. Developers and integrators (a.k.a. release managers) need to comprehend changes before actually merging these changes with the system release or internal development branches. For this, they rely on repositories to extract patches. Patches are changes of source code that do not track the complete history of actions that led to the changes [Robbes 2008a, Ebraert 2008], this is because version control systems do not record every single change that result in a version. From that perspective, operation-based merging [Lippe 1992, Dig 2008] is ruled out since it is based on the idea that either refactorings or every single action made by the programmer is fully recorded.

As mentioned before, the state-of-the-art in industry and open-source development is often limited to good diff tools. Guiffy in Eclipse or Monticello in Smalltalk support a three-way merge (*i.e.,* common ancestor is taken into account to support automatic merging and conflict resolution) [Lindhom 2001]. Diff tools that show the changes as code snippets can be easily used by developers since they can read and understand code fast. However, diff tools do not show the context of a change at large and the view they provide is essentially driven by text constraints.

By proving a characterization of changes [Dragan 2011, Dragan 2010], developers can ease the understanding of these changes [Fritz 2010, Ren 2004] and tackle the problems presented next.

2.3.1.1 Problem 1: Conflicting Changes

Often a developer performs a change against an old version of the system. Two questions then arise: a) What was the delta in the context of the version of the system at the time of the change?, and b) How should that delta be interpreted in the presence of the current version of system?. For example, references to runtime objects of methods in Pharo were managed with the class MethodReference until version 1.4. A method defined in this class for retrieving the name of the method object was methodSymbol. As in Smalltalk a method name is known as *selector*, methodSymbol was later renamed to selector. Integrating changes done to this method in a version where it was still called methodSymbol to a later version where it was already renamed to selector shows a simple case of how conflicting changes may be introduced if they are not interpreted beforehand. The version control system will merge methodSymbol without raising conflicts as in the current version no method named methodSymbol exists.

2.3.1.2 Problem 2: Test Impact

An integrator is often under stress due to the fact that some changes should be integrated whereas at the same time there is no guarantee that no new bugs get introduced or that these changes may not affect the semantics of unchanged code. In many cases integrators rely on regression testing [Elbaum 2000, Elbaum 2003] that is a key aspect especially in presence of complex code for assessing changes. Even though regression testing can assure that old tests work with the new changes, it does not necessarily mean that no semantic errors are introduced. Especially, in systems where not every functionality is accompanied by tests.

In Pharo and in many open-source systems there are no dedicated code test specialists to verify that every single part of the system is covered by a test, and it is also not possible to guarantee that every new change comes with a test suite. Unfortunately, commit policies such as *"Test your changes before committing"* or *"Do not commit changes in the presence of failing tests in the local workspace"* cannot be automatically enforced. The analysis of changes is left to the integrators who need to understand such changes before assessing their impact.

2.3.1.3 Problem 3: Impact of Changes

Changes can be integrated without raising syntactic conflicts, however they might negatively affect the semantics of a system. Understanding the impact of a change is a much more difficult problem. For example, this is particularly complex in the presence of the *yoyo effect* [Wilde 1992,Taenzer 1989] and the *fragile base class problem* [Steyaert 1996]. The problem is that a simple change in a class hierarchy may break existing framework customizations. In such a context the location in the class hierarchy is a first step to assess how many subclasses are impacted by a change and to determine their clients.

Consider a part of the hierarchy for manipulating *collections* in Smalltalk that consists of the class SequenceableCollection which is the superclass of OrderedCollection, which in turn is the superclass of SortedCollection. SequenceableCollection defines the method join:, and both OrderedCollection and SortedCollection override it. Modifying join: in OrderedCollection not only affects this class, but its subclass SortedCollection as well. join: in SortedCollection depends directly of the implementation of join: in OrderedCollection. Therefore, the clients of SortedCollection can be subject to undesirable effects. Again, the integrator must understand these changes to assess their impact on the system.

2.3.2 Merging Support Challenges

While current-day version control systems can automatically merge and resolve *direct merge conflicts*[14], the detection and resolution of conflicts is based on a *textual* analysis. The real challenge lies in taking into account the actual *contents* (*e.g.,* program entities) of the modifications during the merging process. This could help detect *indirect merge conflicts*[15], and unexpected interactions between changes that may cause faulty program behavior, even when conflicts were not reported.

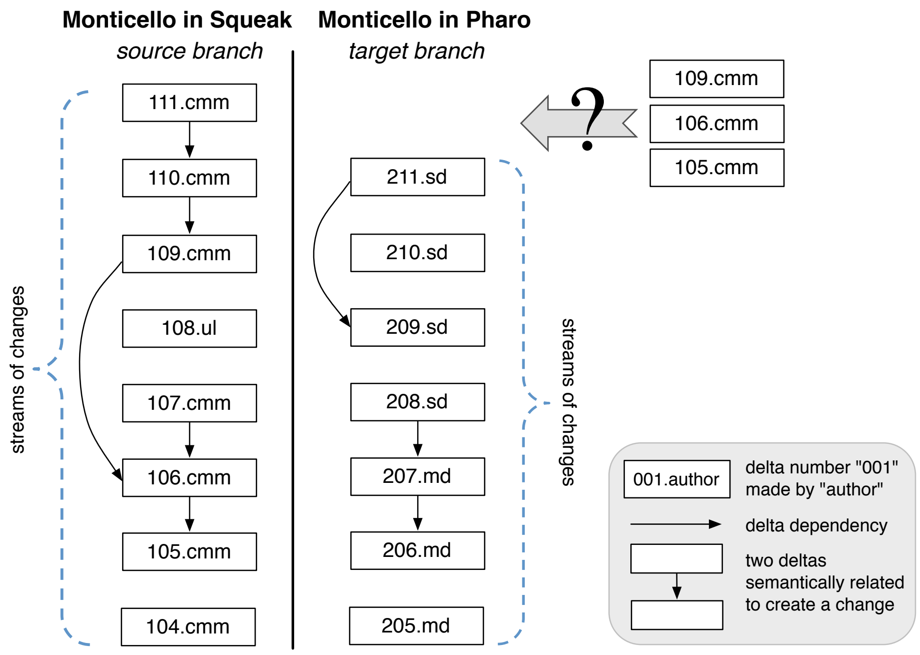


Figure 2.6: *Example: stream of changes of the Monticello system.*

[14]Direct merge conflicts arise when several developers affect concurrently the same file.

[15]Indirect merge conflicts arise when changes made by different developers on different files adversely impact each other.

We illustrate the merge challenges by presenting an example that shows the problems faced by developers that need to merge features across branches.

Figure 2.6 shows two streams (sequences) of changes of the Monticello distributed version control system (core package) in two forks Squeak and Pharo as an example. Here we tagged Monticello in Squeak as the *source branch* and Monticello in Pharo as *target branch*. Note also that each node represents a *delta* (*i.e.,* a set of changes extracted from two versions); directed edges between deltas indicate that one delta depends on another delta; and the numbers of the deltas in the *source branch* are unrelated to the numbers of the deltas in the *target branch*.

Now consider that a developer wants to bring changes from Squeak into Pharo. The developer working on the target branch has to understand the changes that have been performed in the source branch so that he can integrate some of them into the target branch. With current-day tool support, the developer has to navigate the source branch manually to recover the dependencies between the changes. For example, if he wants to apply the delta `109.cmm`, he has to know that he should first apply the deltas `106.cmm` and `105.cmm`, and second, that some part of the changes may conflict with the current target branch. Again he has to identify such problems manually.

2.3.2.1 Problem 1: Textual merge

While version control systems offer support for merging versions, this is mostly limited to a textual merge. Such systems do not take into account semantics of the (object-oriented) programming language used or how the merged changes potentially introduce semantic conflicts. In these cases, it is up to the integrator to analyze the changes manually and assess whether it is feasible to merge these changes, how they impact the branch and how they can be integrated. Even when changes can be merged there is no guarantee that the semantics of the system are correct, or that such merged changes do not have an unintended impact on unchanged code.

2.3.2.2 Problem 2: Cherry picking

The task of merging non-trivial changes between various branches of a software system is still done largely manually. Especially in the case where the branches to be merged have evolved independently and therefore drifted apart, and automatic merging leads to an abundance of conflicts, or where a developer wants to integrate code changes from one branch into another (known as *cherry picking*) explained in previous section.

Over time, it becomes increasingly difficult and tedious for a developer to determine whether a change from another branch can benefit his branch, whether the dependencies of that change exist in his branch, whether the change will introduce bugs into his branch, and how the change relates to any modification he has done in his branch.

2.3.2.3 Problem 3: Identifying change dependencies

Version control systems do not provide developers information about dependencies between changes (*i.e.,* which code is needed by a particular change to be semantically correct). Assessing which changes are needed by a particular one has therefore to be done manually. As an example, consider the case in which a developer wants to introduce the changes from the source branch delta *109.cmm* into the target branch. To do so, the developer has to check all previous changes to find out that delta

109.cmm depends on delta *106.cmm*, which in turns depends on delta *105.cmm*. Therefore these three deltas will probably need to be integrated simultaneously. Other problems left to the developer are assessing the impact of integrating these changes into the target branch and how they can be integrated without breaking the system or without introducing unwanted features.

2.4 Requirements of Solution

Our solution is to assist the integration process by means of tools that can provide developers with information that is needed to perform integration activities. There are several factors that motivate our solution, such as: (a) developers lack semantic information about the changes, (b) developers need to manually extract information (*e.g.,* dependencies between changes) that can guide the integration, (c) developers need to assess the impact of changes before the actual integration or merging.

Providing (semi)-automatic support to assist integration as a means to alleviate the challenges and problems present in the integration of changes, is a complex challenge as well. We do not target fully automated support because human expertise is necessary to understand and assess the impact on the semantics of a system.

We established two conceptual requirements for a solution that can assist the integration of changes in the presence of branches and forks, and one technical requirement for providing such support.

- *Characterizing Changes* can aid developers to comprehend changes. By characterizing changes we can provide relevant information to developers about the changes and their context that will ease the understanding of changes, the identification of the potential impact of integrating the changes, and therefore will help developers taking decisions about the integration process in general. This requirement is further described in Chapter 3.

- *Characterizing Stream of Changes* can guide developers to merge changes. By establishing the dependencies of a change within a stream of changes (*i.e.,* within a branch/fork) we can characterize that change within the stream, and therefore aid developers to understand changes within the stream, to cherry pick changes and to identify which dependencies present in the stream are needed. This requirement is further described in Chapter 3 as well.

- *Infrastructure* is needed at a technical level to be able to characterize changes and streams of changes. We need to establish the adequate infrastructure that allows us to represent the evolution of a system, the changes and their dependencies in a way that we can efficiently infer semantical information to guide the integration process. Moreover, such infrastructure should be the foundation for building tool support. We present our infrastructure in Chapters 4 and 6.

2.5 Conclusion

In this chapter we have presented that collaborative software development is widely used mainly for large systems. The fact that it allows a team of developers to work independently of each other in multiple tasks of the same project without restrictions on time and location is a powerful approach applicable not only for open-source development but also for proprietary source development.

Version control systems support collaborative development by integrating approaches such as branching and merging, in particular distributed version control systems (*e.g.,* Git, Mercurial) that are mainly focused on supporting distributed development have given branching a central role to guide the development process.

We have presented a concrete example of an open-source development project – Pharo – in the Smalltalk community that represents a large system used by both the academia and industry. Independently of the programming language or version control system used in Pharo, it really shows how branching and merging plays a fundamental role in its development. The Pharo ecosystem is complex and involves internal and cross branching.

We emphasized that the big problem inherent in the use of branching is the integration of changes. Integration is not easy, and integration is not one single task, it involves complex activities such as understanding changes, selecting changes and their requirements, determining the impact of changes, and merging changes. Moreover, the lack of support for performing these activities has resulted in a process that in many cases is done manually. Even though, version control systems support merging, the ones that are widely used do so at a textual level, the semantics behind the changes needs to be understood by developers in order to take decisions in each of the activities involved in the integration process.

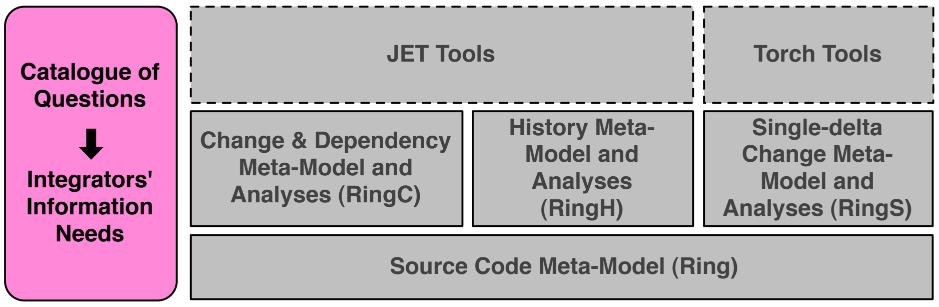
Supporting integration of changes is a difficult task, there are several challenges that need to be taken into account to propose the requirements for an adequate solution. We summarized these challenges in two groups: a) change characterization, and b) merging support. In each group, there are several problems that developers face when integrating changes and that are related to the activities mentioned before. By providing a characterization of changes and a characterization of stream of changes we can provide a solution that aids developers to tackle these challenges and support the integration process. Along with the characterizations we require the right infrastructure to build tool support needed by developers.

The next chapter presents the facets of changes, in concrete, we introduce terminology used in the integration context, questions that are raised by developers when integrating changes, and we present the characterization of changes and stream of changes needed to guide developers integrating changes.

CHAPTER 3
Facets of Changes

Contents

Contributions Map



Catalogue of Questions
Integrators' Information Needs
JET Tools
Torch Tools
Change & Dependency Meta-Model and Analyses (RingC)
History Meta-Model and Analyses (RingH)
Single-delta Change Meta-Model and Analyses (RingS)
Source Code Meta-Model (Ring)

Overview

This chapter covers four topics that complement the background and problems associated with the integration of changes in a collaborative development process, as presented in Chapter 2. First, we present an overview of the integration process to introduce the definitions and terminology that we use in this dissertation. Second, we present a catalogue of questions that are raised during the integration of changes as a means to identify the integrators' information needs to answer these questions. Third, we describe and characterize the different kinds of information that can be used to answer the integrator questions. These kinds of information enable us to provide a characterization of changes and streams of changes as part of our solution to assist integrators. Fourth, we study the state-of-the-art relating to this dissertation. Our goal is to investigate how other approaches represent and provide these kinds of information, and which kinds of tools exist nowadays to support answering developers or integrators questions.

3.1 Integration Process: Overview

In Chapter 2 we explained the collaborative software development process and how version control systems support this process by enabling branching and merging. In this section, we illustrate the conceptual process of integrating changes as it can be found in open-source projects in order to introduce the definitions and terminology that we use in this dissertation.

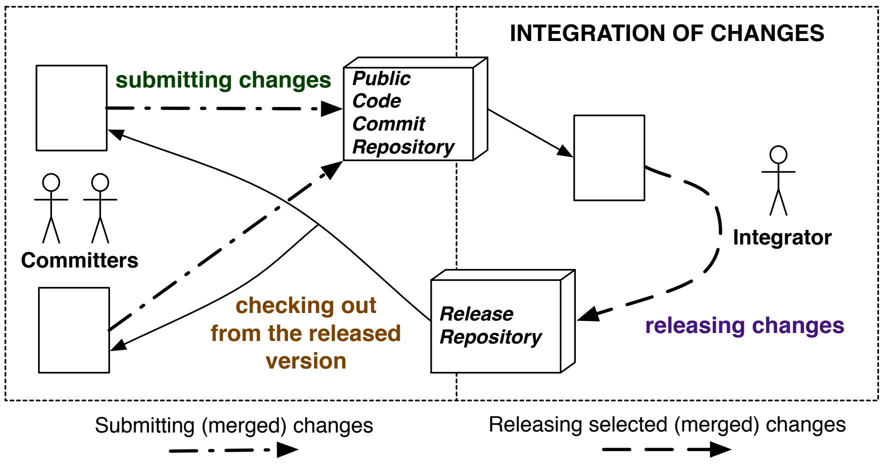

INTEGRATION OF CHANGES
submitting changes
Public Code Commit Repository
Committers
Integrator
checking out from the released version
Release Repository
releasing changes
Submitting (merged) changes
Releasing selected (merged) changes

Figure 3.1: *Integration process: different roles and actions.*

An overview of the flow of changes and their integration is shown in Figure 3.1. Here we focus on the groups of developers that support the production of a new release. From this perspective, and ignoring issues related to testing (acceptance testing and others), we can identify two roles for developers: *committers* of changes and *integrators* of such changes.

- ***committers*** checkout code from a repository, be it the main branch or any other (*e.g.,* development, feature, alpha or beta version) branch. They work on fixing bugs, enhancements or new features and submit (*i.e.,* publish) their changes to a repository.
- ***integrators*** analyze the code of *committers*, merge selected changes (which were made to previous versions) and release them into the current version.

The process of submitting changes to a shared repository is usually regulated by a commit policy that committers are encouraged to follow, such as the *commit early and commit often* rule [Berlin 2006] (*e.g.,* continuous integration[1]), that aims to minimize merge conflicts, avoid duplicate development, and avoid test failures. However, commit policies are generally not automatically enforceable, and integrators have to deal which such potential problems.

The **integration** of changes covers the complete process of gathering changes from several committers and including them in a version of the system. This process involves several activities such as understanding changes, selecting (cherry picking) changes, *merging* changes, resolving merge conflicts, compiling, and testing the merged code. **Merging** is a part of the integration process, that represents the technical operation in which two sets of changes are applied to a file or set of files. The changes come from different branches or forks, and may raise conflicts when they affect the same code in different ways.

[1]Continuous integration: http://www.martinfowler.com/articles/continuousIntegration.html

3.2 Definitions and Terminology

In this section, we introduce the definitions and terminology that we use in the remainder of this dissertation. Not all of them are well-known or standard terminology in the context of integration of software changes, therefore we define them explicitly for easing the understanding of our work. Even though these definitions and terminology are applicable for any object-oriented programming language and version control system, we present them in the context of the Pharo Smalltalk platform and the Monticello distributed version control system, which are part of the technology used to implement and illustrate our approach.

Within the definitions we use the notation KEY_{name} to refer to several terms. Classes C_{Test}, methods M_{run}, attributes A_{name}, changes CH_m, deltas D_n, or snapshots S_o.

Program entities and relationships. The key *program entities* that can be found in an object-oriented software system and that we take into account in our approach are *packages*, *classes*, *traits*, *methods* and *attributes*. The *relationships* between program entities that we take into account are *attribute accesses* (*e.g.,* attribute A_{value} is being read and written within the method M_{foo}), *method calls* (*e.g.,* method M_{foo} invokes method M_{bar}), *class references* (*e.g.,* method M_{foo} refers to class C_{Zoo}) and *class inheritance* (*e.g.,* class C_{Lion} inherits from class C_{Animal}).

Repositories. A *repository* is where the source code and other artifacts of a system are stored. Therefore, it contains the evolution and changes of such a system. How the data is stored depends on the version control system or application managing the repository. For example, in Monticello – the distributed version control system supported by our contribution – each package is published individually containing not only changes but all the program entities existing in the package at that point in time.

Commits. A *commit* in the context of version control systems refers to the act of submitting changes of the source code to the repository. A *commit* can also refer to the group of additions, modifications and removals made to the source code that developers submit to the repository and result in a new revision (also known as version). However, in Monticello a commit as a whole does not exist, because changes are submitted to their respective packages. For example, if a set of changes (commit) cross-cuts three packages, such changes are submitted separately resulting in three *package versions*. Monticello identifies changes that happened in a particular package version, by comparing that version to its previous version.

Snapshots. A *snapshot* is a set of program entities and relationships at a given point in time in the history of a system. This set of entities represents the complete system under analysis, in contrast to commits that refer to the changes submitted at a point in time. A snapshot is derived from a commit. However, it also includes unchanged program entities and relationships present in the history at that point in time. In Monticello, to determine which versions of the packages belong together (*i.e.,* represent a commit), we use a sliding window technique [Zimmermann 2004a] that considers that multiple packages belong to the same commit if they are committed by the same author within a time interval of 5 minutes.

History. A system's *history* encapsulates the evolution of its program entities and relationships. We consider a system's *history* to be a *graph of snapshots*, where each node in the graph (*i.e.,* a snapshot) is connected to its predecessor(s) (previous snapshots of the system) and successor(s) (subsequent snapshots of the system).

Changes. A *change* is any alteration applied to a program entity. We distinguish between three kinds of actions applied by developers, namely removals, additions and modifications. For example, introducing the new class C_{Lion} results in the *added change* to C_{Lion}, modifying the method M_{foo} results in the *modified change* to M_{foo}, and so on.

Deltas. A *delta* is a set of changes representing the differences between two successive *commits* or *snapshots* (known as snapshots S_{base} and S_{target}) present in the history. Note that in the context of Monticello we also use *delta* to represent the differences between two *package versions*. A delta can be considered equivalent to applying a *diff* to two versions of a file and resulting in textual differences.

Stream of changes. A *stream of changes* is a sequence of successive commits in the history of a system. In other words, a stream of changes is a sequence of deltas that may represent a part or the whole evolution of a system. That sequence of deltas is considered to be a graph of deltas, where each delta is connected to its predecessor(s) (previous deltas) and successor(s) (subsequent deltas). For example, from the history of Pharo consider the feature "RecentSubmissions" (which handles the recent changes loaded into the system). In Pharo 1.4 the revision of this feature dated August 12th 2011 was released. Up until now (June 21st 2012) this feature includes 58 revisions in the *develop* branch that may be considered for the next release Pharo 2.0. In this case, these 58 revisions developed in the past 10 months represents a *stream of changes* for that feature.

Change dependencies. A *change dependency* captures the fact that a given change CH_y potentially depends on another change CH_x (*i.e.,* $CH_y \rightarrow CH_x$). For example, if a modification to method M_{foo} adds a call to a new method M_{bar}, this change introduces a *change dependency* of M_{foo} to M_{bar}. That means that in order to integrate the modified method M_{foo}, the added method M_{bar} is needed. Such a dependency can exist between changes within the same delta or between changes in different deltas.

Delta dependencies. A *delta dependency* expresses a dependency from delta D_n to delta D_m (*i.e.,* $D_n \rightarrow D_m$), where a change CH_y in D_n depends on a change CH_x in D_m (*i.e.,* the change dependency $CH_y \rightarrow CH_x$ exists). That means that a delta depends on another delta if any change within it depends on a change in other delta. Considering the example presented in the definition of change dependencies but assuming that the method M_{bar} was added in delta D_7 and that the method M_{foo} was modified in delta D_8, then due to the change dependency between both changes, the *delta dependency* $D_8 \rightarrow D_7$ is introduced.

3.3 Questions Integrators Ask

In this section, we present a catalogue of questions that integrators ask when performing integration of changes. The main motivation behind obtaining these questions is to identify and understand which are the information needs of developers that deal with integration activities. As we mentioned before, integrating changes is a difficult and tedious process, since integrators can be not the people who implemented the original software nor the ones who changed it. By knowing which real questions are raised during integration and are troublesome to answer, the complexity of this process can be understood, common factors can be extracted to characterize changes, and future solutions can be assessed.

Before presenting details of how we obtained the questions, we present a brief background (extended in Section 3.5.5) that justifies the usage of questions as a means to support development activities.

In literature, we find a number of studies that have identified the set of questions that software developers ask themselves when dealing with changes [Sillito 2006, Sillito 2008, Fritz 2010, LaToza 2010]. Sillito *et al.* [Sillito 2006, Sillito 2008] studied the information a developer needs to know about a code base while *performing change tasks* and how developers go about discovering that information. They propose 44 questions that developers asked themselves during development. However, many questions are related to code navigation ("Where is this method called or type referenced?" or "What are the arguments to this function?"). Fritz and Murphy [Fritz 2010] focused on team related information about changes. They listed 78 questions raised by developers mainly when concerned about the impact of their work on other team members. Out of these 78 questions, 35 questions are code specific, for example, they included questions such as "Who is using that API [that I am about to change]?" or open-ended questions such as "What is the evolution of the code?". LaToza and Myers [LaToza 2010] gathered *hard-to-answer* questions about code in general from professional developers. They obtained a broad set of 371 questions, of which 218 questions are about changes. Such questions are classified in 10 categories such as debugging, policies, history, refactorings, teammates, building and branching, etc. Few of these categories (*e.g.,* history, building and branching) contain questions such as "How can I move this code to this branch?" or "Have changes in another branch been integrated into this branch?" that can complement our findings.

Even if previous work lists a large set of questions, many of these questions are not specific enough or do not apply to the context of integrating streams of changes. To assess if our contributions can answer questions adequately, we needed to identify the specific questions developers ask when integrating changes. Therefore, we complemented these studies with our study discussed next.

3.3.1 Methodology

We conducted an open call to the developers of three Smalltalk communities to compile the questions. Specifically, we sent a mail to three mailing-lists (VisualWorks users mailing-list *vwnc@cs.uiuc.edu*, ESUG (European Smalltalk User Group) *esug-list@lists.esug.org*, and Pharo project mailing-list *Pharo-project@lists.gforge.inria.fr*) requesting input on the questions they ask themselves when integrating. We first provided an overview of the reasons of our study, next we asked "What are the questions that you ask yourselves when you are merging (or want to merge) changes into your projects?", finally we added six typical questions raised by one of the main Pharo integrators (*e.g.,* "Is this change impacted by a change that happened in another branch of my software?") as a way to guide their answers.

In a period of 10 days we received the responses of 20 developers who integrate changes on separate small, medium and large sized Smalltalk projects. The answers were diverse among the group. (a) 8 participants provided concrete questions. (b) 9 participants provided concrete questions and extra feedback (merge situations they deal with, policies they follow when merging, explanations of why they ask these questions or think they are challenging, and broad ideas for tools supporting merging). (c) 3 participants did not list any question at all but rather included general feedback regarding their desiderata for merging tool support. This information was analyzed and yielded 56 questions. Moreover, we took into account the studies presented above and extended our findings with 8 questions taken from these studies. Finally, Stéphane Ducasse, a Pharo integrator helped refining and verifying the questions.

The resulting catalogue is composed of 64 questions that we clustered into 5 different categories: (a) authorship/ownership, (b) change nature, (c) structural change characterization, (d) bug tracking infrastructure, and (e) changes within a stream.

Threats to Validity. Our study was performed by an heterogenous group of participants from both industry (majority) and academia (minority but supporting real projects).

The fact that all the participants are Smalltalk experts, and their answers are focused on the integration of changes in the context of Smalltalk projects may be considered a threat to validity of our study. However, our audience was quite diverse, the participants work on different Smalltalk projects which follow different development policies. This is a positive aspect because we received different points of view regarding the integration process.

Furthermore, this study covers a topic that is present in any collaborative development process independently of the programming language and infrastructure used. The related work discussed before also proposed the identification of developers' information needs by means of questions. Even though they covered other broad aspects, there are questions overlapping with our findings which reveals common information needs. This shows the generalizability of this kind of studies, and the questions gathered in our study can be used to assess future solutions developed with other platforms.

3.3.2 Catalogue of Questions

We briefly describe each category prior to introducing its respective questions. Each question is accompanied by an identifier (*e.g.,* A_1) that it is used to refer to the question in later sections. Note also that we add references to the questions that are similar to questions (or were identified) in any of these studies [Fritz 2010, LaToza 2010, Sillito 2008].

Authorship/Ownership. The first group of questions is related to the owner of the original code, author of the changes and committer. These questions reflect the implicit quality of the committer and level of reliability of such changes.

	Author/Owner questions
A_1	Who wrote the original code that was changed?

Continued on Next Page…

	Author/Owner questions
A_2	Who is the owner for this changed code? (part of the catalogue of Fritz and Murphy/LaToza and Myers)
A_3	Has my code been changed?
A_4	Who made this change? (also part of the catalogue of Fritz and Murphy)
A_5	What is the general quality of the change committer?
A_6	How many people have contributed to this sequence of changes?

Change nature. The second group of questions is related to the nature, behavior and intent of a change. Such questions can be mostly applied to changes within a single delta. Note that some of these questions are open-ended and therefore inherently difficult to answer automatically. Moreover they may require up front knowledge of the system as well.

	Behavioral questions
B_1	Does this change improve the quality?
B_2	Is this change correct? (also part of the catalogue of LaToza and Myers)
B_3	What is the reason for this change? (also part of the catalogue of Fritz and Murphy/LaToza and Myers)
B_4	Was this change intentional, accidental, or a hack? (part of the catalogue of LaToza and Myers)
B_5	What kind of change is it? (Bugfix/New feature/Refactoring/Documentation)
B_6	What is the total impact of this change? (part of the catalogue of Sillito *et al.*)
B_7	What will be (or has been) the direct impact of this change? (part of the catalogue of Sillito *et al.*)
B_8	Did this change fix/break tests? Which tests?
B_9	Did the tests work before the changes?
B_{10}	How can I test this change? (part of the catalogue of LaToza and Myers)
B_{11}	Is the change covered by tests? What is the coverage? (also part of the catalogue of LaToza and Myers)
B_{12}	If I just apply the change, what are the parts of my current system that it will break?
B_{13}	If the merge succeeds, will the change work later?
B_{14}	What are the implications of this change for API clients? (also part of the catalogue of LaToza and Myers)

Structural change characterization. The third group of questions is related to the structure of the original code as well as the changes. They cover various aspects in terms of volume, impact volume, dependencies (which packages, classes should be loaded before), and so on. From that perspective, they are not tailored to a stream of changes but more to a single delta.

	Structural change characterization questions
S_1	How large is the change?
S_2	What is the scope of this change? (which/how many classes/packages/..., is local/global?)

Continued on Next Page...

	Structural change characterization questions
S_3	Is this change confined to a single package?
S_4	What is the complexity of the changes/of the touched entities?
S_5	Does this change define only one feature?
S_6	Can we split this change?
S_7	Do all the changes within the commit belong together or are they unrelated?
S_8	Are there other packages that would need to change as well to incorporate this change?
S_9	When multiple packages are committed at the same time, do I really need to load all of them now, or can I just load/merge with the version of the package I am working on?
S_{10}	Can I apply this change?
S_{11}	What are the required structural dependencies? (also part of the catalogue of LaToza and Myers)
S_{12}	What other changes depend on this change? (also part of the catalogue of LaToza and Myers)
S_{13}	What are the conflicts?
S_{14}	What parts of the system are directly using the changed behavior?
S_{15}	Which features/classes/methods have been changing? (also part of the catalogue of Fritz and Murphy/LaToza and Myers)
S_{16}	Is this change impacted by a change that happened in another branch of my software?
S_{17}	Does the change follow rule checking/conventions?
S_{18}	Is the vocabulary used in the change consistent with the one of the system?
S_{19}	What code is related to a change? (part of the catalogue of Fritz and Murphy)

Bug tracking infrastructure. The fourth group of questions is related to bug tracking facilities.

	Infrastructure questions
I_1	To which bug entry does this change relate?
I_2	Have other bugs related to the change been reported?

Changes within a stream. The final group of questions is related to situating changes within the context of a stream of changes, as well as to the time at which the change occurs. In particular when working on a stream of changes, these questions capture the place of a change within the stream.

	Temporal and change stream questions
T_1	When was this change made? (also part of the catalogue of LaToza and Myers)
T_2	What is the whole history of this method/feature? (also part of the catalogue of Fritz and Murphy/LaToza and Myers)
T_3	In which version this method change?
T_4	Did this method/feature change (a lot) recently/in the past? (also part of the catalogue of LaToza and Myers)
T_5	Did this change ever happen before?
T_6	Is this change still the most recent one?

Continued on Next Page…

	Temporal and change stream questions
T_7	Is there any pending change in the sequence that supersedes it?
T_8	Is this change part of a whole series of changes?
T_9	Does this change depend on previous ones?
T_{10}	Is the change ever used in subsequent changes?
T_{11}	Is this change reverting the code to an old state?
T_{12}	What else changed when this code was introduced or changed? (also part of the catalogue of LaToza and Myers)
T_{13}	What are the changes made by the same authors/during the same time period?
T_{14}	Did the same methods/classes change together in a particular version? what were the missing changes?
T_{15}	Was this method/class renamed in the past? in which version?
T_{16}	What were the senders of this method in a particular version?
T_{17}	What are the current senders of this method in a particular version?
T_{18}	What are the current message calls by this method in a particular version?
T_{19}	What are the changes based on the latest common ancestor between the version of the changes and the system?
T_{20}	Have changes in another branch been integrated into this branch? (part of the catalogue of LaToza and Myers)
T_{21}	How can I move this change to this branch? (part of the catalogue of LaToza and Myers)
T_{22}	What are the (previous) changes needed to merge this change with another branch? (also part of the catalogue of Sillito *et al.*)
T_{23}	What are the clients potentially impacted by this change in the destination branch/fork?

3.3.3 Answering Integrator Questions

The complexity of the integration activities as a whole process is reflected by the lack of adequate support for developers that need to integrate changes. The state-of-the-art includes several studies that show the importance of gathering the questions that developers ask when performing diverse development activities. The common goal of these studies is to identify and understand the developers' information needs to reveal opportunities for developing new languages and tools that make answering these questions easier. That information can support activities such as understanding changes, assessing the impact of changes, cherry picking changes, navigating changes, identifying team related involvement in changes, supporting decision making for branching and merging, and so on.

With version control systems (discussed in Chapter 2.1.1) and other approaches (discussed in Section 3.5) it is impossible to answer all these questions. In particular there are no straightforward answers to questions related to the nature of the changes (*e.g.,* "Is this change correct?" or "What is the reason for this change?") that involve the semantics of the system and therefore require user intervention. Answering many of these questions can potentially be time consuming and error prone. Advanced low-level tools are needed. The state-of-the-art discussed in Section 3.5 shows several approaches dedicated to assess the impact of changes. Still these approaches are semi-automatic as they require users to provide input, interpret the output generated by tools, and ultimately decide what to do which such output. However, activities such as understanding changes and mainly *cherry* *picking* have been ignored by widely used tools. For example, well known version control systems (*e.g.,* SVN, Git) are unable to answer questions such as “What are the required (structural) dependencies?” or “What other changes depend on this change?” as they merely process changed code as text. Developers have to manually process the changes before the actual merge happens, and even though these changes may be successfully merged, there is no guarantee that the functionality of the system is 100% correct or that no ripple-effects will appear in the future due to prior changes.

We do not claim that tools should support activities like cherry picking or change understanding fully automatically, but providing semi-automatic support will greatly enhance the productivity of integrators. Our contributions are towards that direction, and we make use of the questions presented in this section to identify the key information to characterize changes and streams of changes. These characterizations enable us to *assist the integration* process.

3.4 Information Needs for Change Characterization

Based on the catalogue of questions presented in Section 3.3.2 we identify several kinds of information that can help us to accomplish the characterization of changes and streams of changes.

We have classified such information in four categories: (a) descriptive information, (b) structural information, (c) semantic information, and (d) historical information. While the last category includes information related to changes in the context of a stream of changes, the other categories provide information that can be used for characterizing changes within a single delta and within a stream.

In the following, we present an analysis of this information that serves to define the requirements for our solution introduced in Chapter 2.4.

3.4.1 Descriptive Information

- *Size*. Characterizing a change in terms of its size gives a first impression of a change. A common metric used to measure the size of a system is based on the number of lines of code (LOC). In the same way, a first measure to establish the size of the changes is in terms of lines of code impacted. This can be used to answer questions like “How large is the change?”. Note however, that the size of a change is only indicative since a small change can have huge effects. Still, the size in terms of lines of code can indicate how difficult and risky an integration of such changes will be. Combining the size of changes with the number of changes within a delta can be used to characterize changes and establish patterns to aid in understanding changes (*e.g.,* replacing a method call is represented by multiple changes each impacting two lines of code[2]).

- *Author/Owner*. Knowing *who* changed a piece of code (*i.e.,* author) and *who* committed the changed code to a source code repository (potential owner) is relevant, especially in a team development process. Note that who changed the code may not be the same developer who committed the changes, or that who committed may not be the owner of a changed feature. The author of a change and the committer are kept for tracking purposes. This allows us to answer questions such as “Who made this change?”, “How many people have contributed to this sequence of changes?” or “Who wrote the original code that was changed?”. To answer questions such as “Who is the owner for this changed code?” requires the prior identification of the

[2]In Section 5.6 we revisit a concrete example for this scenario.

owner. The owner(s) of a feature can be derived by metrics considering the number of changes or size of changes that a developer has committed for a particular feature. Identifying the author and committer of a change can aid other developers to establish the degree of reliability of such a change. Managers and integrators can watch out for code which is contributed by developers who have inadequate relevant prior experience, as their changes could lead to more failures in the system. Ownership complements authorship, changes made by the owner of a feature could be considered safer that changes made by other developers [Bird 2011]. Integrators can use the author and the owner information as factor to assess changes.

- *Time*. The point in *time* at which a change happened (*when*) is key to establish the order of a change, and helps to identify sequences of changes during the evolution of a system. IDEs may record the time when a source code change is made, and version control systems register the time when changes are submitted to the repository. Questions such as "When was this change made?" or "What are the changes made by the same authors/during the same time period?" can be answered. Temporal information is heavily used by processes and analyses of the source code to support several activities in the development process (*e.g.,* change impact analyses [Chesley 2005, German 2009, Herzig 2011]). In our context, time is useful to establish the history of a system and dependencies of changes. Therefore, questions such as "Is this change still the most recent one?" can be answered.

3.4.2 Structural Information

- *Structure*. The packages, classes and methods are the core of programs and can be used by the tools. Identifying which entities changed and how they relate to each other can ease in understanding these changes (*e.g.,* a pull up/push down method refactoring can be detected by identifying if their respective classes belong to the same class hierarchy). This information can support answering questions such as "Do all the changes within the commit belong together or are they unrelated?" or "Does this change define only one feature?". The number of packages, classes and methods compared to the application size is another simple characterization of a change (*e.g.,* "What is the complexity of the changes/of the touched entities?"). However, such an estimate can be misleading, for example when a simple *API use* change is applied in the complete system.

- *Change Scope*. Identifying if the changed source code is *local* to a method, class, hierarchy, or package, or that it *cross-cuts* multiple entities establishes the scope of a change (*e.g.,* multiple changes are contained in one single package). Questions such as "What is the scope of this change? (which/how many classes/packages/..., is local/global?)" or "Is this change confined to a single package?" can be answered. Moreover, assessing a local change is often simpler than one cross-cutting several hierarchies or packages. Cross-cutting changes that are not necessarily structurally related but evolve together may indicate a coupling between these changes ("Do all the changes within the commit belong together or are they unrelated?"). Knowing this information may help identify missing changes. Therefore, getting a fast overview of the location of changed program entities in the context of the hierarchy and package structure is important to assess changes.

- *Kind of Actions.* Understanding whether the changes are mainly *adding*, *removing* or *modifying* behavior is another level of characterization. Whether changes are at the level of entire methods (*i.e.,* a method was added or removed) or intra method (*i.e.,* a method's body was modified) is another element. Whether the changes were actually changing the semantics of the system (*e.g.,* not just changes to license or comments) is complementary to the other information. This can guide the answering of questions such as "What kind of change is it? (Bugfix/New feature/Refactoring/Documentation)". Identifying specific actions such as renamings[3] would definitely improve any characterization of changes. However, this can be a challenging process when the addition differs a lot from the removal.

- *Kind of Entities.* Characterizing changes by kind of entity (*e.g.,* fields, methods, comments, etc.) they affect is straightforward. This can be used to answer questions such as "Which features/classes/methods have been changing?". This, combined with previous characterizations such as *kind of actions*, can ease in assessing the impact of changes. For example, if only comments were affected at class or method level, developers can rapidly identify that these changes have no semantical impact on the system and answer questions like "What parts of the system are directly using the changed behavior?". Therefore, they can integrate these changes without dedicating time to understand and assess their impact.

3.4.3 Semantic Information

Vocabulary. Identifiers (class names, field names, function names and parameter names) and comments are important elements of the source code that give hints about semantics and intent of the developers [Takang 1996, Anquetil 1998]. They represent the vocabulary present in a system, and such vocabulary is affected when the system evolves [Abebe 2009] (*e.g.,* if new features are introduced or if existing behavior is removed is reflected as additions and removals of identifiers). Assessing the difference in vocabulary between a change and its application can give information about whether or not that change fits the existing application. Questions related to the vocabulary such as "Is the vocabulary used in the change consistent with the one of the system?" or "Does the change follow conventions?" can be answered. Moreover, the amount of introduced or removed vocabulary can provide an overview of the changes and their impact. For example, having a large number of changes with a limited number of affected vocabulary (one added and one remove identifier) could mean that a function was renamed, or that a call was replaced among its clients[4]. Or, having multiple changes that do not affect the identifiers (*i.e.,* only comments) indicates that they do not impact the semantics of the system.

Reason. A system is constantly evolving due to fixes of bugs, enhancements, new features or adaptations for changing environment. The reason behind *why* a piece of code changed is fundamental to aid in understanding and assessing the impact of a change. Questions related to the reason of a change such as "What is the reason of this change?" or "Was this change intentional, accidental, or a hack?" are the most difficult to answer only with tool support. Committers can add a description

[3]A renaming is usually stored as a removal and an addition.

[4]In Section 5.6 we revisit a concrete example for this scenario.

(*i.e.,* commit message) about the changes they submit into the repository. Unfortunately, committers may omit important details of *what* and *why* changed, even more when they submit multiple unrelated changes in one commit. Despite this, such information can partially aid integrators during the integration process to answer questions such as “What kind of change is it? (Bugfix/New feature/Refactoring/Documentation)”.

3.4.4 Historical Information

History. The history of a system contains a wealth of information that can be used to understand the system and its evolution, to detect problems in the system, to predict future problems, and so on. In our context, an adequate representation of the history (*i.e.,* not just files) can be used by integrators to understand changes and their potential impact. Therefore it can support answering questions such as “What is the whole history of this method/feature?” or “Did this method/feature change (a lot) recently/in the past?”. Understanding changes within a delta is less complicated than understanding changes within a stream of changes. For this, the context (*i.e.,* history) in which these changes happened is needed. By using the history questions such as “What were the senders of this method in a particular version?” can be supported.

Change Dependency. A specific change can require several other prior changes. This is more relevant in the case when changes come from different branches or forks. For example, if class `C` subclasses class `B`, then class `C` *depends on* class `B`. If the branch in which class `C` is intended to be integrated does not contain class `B`, then the change adding class `C` must be integrated with its dependency (*i.e.,* change adding or modifying class `B`). To support the integration of changes from one branch into another – *cherry picking* – it is fundamental to establish dependencies between changes. Such dependencies can be used to characterize changes and deltas within the stream, and partition changes that should be integrated. The characterization of deltas can guide integrators to prioritize changes, for example deltas that do not depend on any prior change can be tagged as the easy ones. Therefore the integrator could first concentrate on the complex cases, or vice-versa. Moreover, by means of dependencies it can be possible to establish which entities have been changing together. This can be key in spotting problems with a change, and help integrators assessing these changes. Dependencies can support answering questions such as “Does this change depend on previous ones?”, “What are the required structural dependencies?”, “What other changes depend on this change?”, or “Can we split this change?”.

3.4.5 Summary

The identified information is fundamental to support answering many of the questions presented in Section 3.3.2 that integrators ask themselves. We illustrate in Table 3.6 and Table 3.7 two concrete analyses. Table 3.6 shows the questions that can be supported by a particular kind of information. Note that we make use of the questions’ identifiers to refer to the questions. Moreover, a question’s identifier is followed by an asterisk to indicate that such question is partially supported by a kind of information. Table 3.7 shows the extent of the support reached in terms of the number of questions that can be answered. For each category of questions we indicate the number of questions that can be *partially* or *fully* supported.

Kind of Information	**Supported Questions**
Descriptive Information	
Size	A_2*, S_1, S_4, T_4*
Author/Owner	A_1, A_2*, A_3, A_4, A_5*, A_6, S_6, S_7*, T_{13}
Time	T_1, T_2*, T_6, T_{13}
Structural Information	
Structure	B_5*, S_2, S_3, S_4*, S_5*, S_6*, S_7*, S_{11}*, S_{12}*, S_{13}*, S_{15}, S_{19}, T_{12}, T_{13}*
Change Scope	B_5*, S_2, S_3, S_4*, S_5*, S_6*, S_7*, S_{19}
Kind of Actions	B_5*, B_6*, B_7*, B_{14}*, S_5*, S_6*, S_7*, S_8*, S_{11}*, S_{12}*, S_{13}*, S_{14}*, S_{15}, S_{16}*, S_{19}, T_2*, T_3*, T_4*, T_5*, T_7*, T_8*, T_9*, T_{10}*, T_{11}*, T_{12}, T_{13}*, T_{14}*, T_{15}*, T_{19}*, T_{20}*, T_{22}*, T_{23}*
Kind of Entities	B_5*, B_6*, B_7*, B_{14}*, S_1*, S_2, S_3, S_4*, S_5*, S_6*, S_7*, S_8*, S_{11}*, S_{12}*, S_{13}*, S_{14}*, S_{15}, S_{16}*, S_{19}, T_2*, T_3*, T_4*, T_5*, T_7*, T_8*, T_9*, T_{10}*, T_{12}, T_{13}*, T_{14}*, T_{15}*, T_{16}*, T_{17}*, T_{18}*, T_{19}*, T_{22}*, T_{23}*
Semantic Information	
Vocabulary	S_4*, S_5*, S_7*, S_{17}*, S_{18}
Reason	B_3*, B_4*, B_5*, S_5*, S_6*, S_7*, I_1*
Historical Information	
History	A_1, A_2, A_3, A_5*, A_6, S_{12}, S_{14}, S_{15}, S_{16}*, S_{19}, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{13}, T_{14}, T_{15}*, T_{16}, T_{17}, T_{18}, T_{19}*, T_{20}, T_{21}, T_{22}, T_{23}
Change Dependencies	B_6*, B_7*, B_{14}*, S_6, S_7*, S_4*, S_8*, S_9, S_{10}*, S_{11}, S_{12}, S_{13}*, S_{14}*, S_{16}*, S_{19}, T_8*, T_9, T_{12}*, T_{20}*, T_{21}, T_{22}

Table 3.6: *Supported questions by kind of information (* information partially supports answering a question).*

Category of questions (id #questions)	**Partially supports**	**Fully supports**	**Supported questions**
Authorship/Ownership (A_n 6)	1	5	*6*
Change nature (B_n 14)	6	0	*6*
Structural change characterization (S_n 19)	6	13	*19*
Bug tracking infrastructure (I_n 2)	1	0	*1*
Changes within a stream (T_n 23)	1	21	*22*
Supported questions / 64 questions	*15 / 64*	*39 / 64*	*54 / 64*

Table 3.7: *Number of supported questions by category of questions*

We distinguish the amount of support that can be provided (*i.e., partial* and *full*). Easy questions can be answered by means of a single kind of information. For example, A_4 ("Who made this change?") can be answered only by knowing the *author* of the change. Therefore, we say that *author* fully supports the answering of A_4. However, many of the questions are challenging and they cannot be answered using one single kind of information, but with a combination of them. For example, S_7 ("Do all the changes within the commit belong together or are they unrelated?") can be answered by using structural information, semantic information, author, and change dependencies. In this case, Table 3.6 shows in each of these kinds of information S_7*. Meaning that such kind of information partially supports the answering of S_7.

The numbers shown in Table 3.7 are encouraging. 39 out of 64 questions can be fully answered using the identified information, and 15 out of 64 can be partially answered. Note that the 10 remaining questions are in the categories *change nature*, *bug tracking infrastructure* and *changes within a stream*. They are related to testing, change impact and bug tracking. For our current contributions, we have not considered the use of tests or bug tracking information. Taking such information into account is considered future work.

3.5 State-of-the-Art

In this section we introduce the related work that is pertinent to our contributions. We mainly present approaches that are relevant to us because they provide support for similar, overlapping or complementary activities within our context. Such approaches concern modeling source code, history and changes, merging, change impact analysis, change dependencies, and understanding development tasks.

For each topic, we present several approaches that are related or relevant to our context. A discussion about these approaches and how their purposes differ from ours is included at the end of each topic. Moreover, we also provide a brief analysis of other related work at the end of this section.

3.5.1 Modeling Source Code, History and Changes

To provide support for answering the questions that integrators raise during the integration of changes, we need to model the necessary information presented in the previous section.

The use of meta-models as a means to provide common representation frameworks that can be leveraged by various software engineering tools is not new. Along the representations of object-oriented programs, several approaches exist that model changes and multiple versions of a system. A good overview of the early research in this area can be found in the book chapter by D'Ambros *et al.* [D'Ambros 2008].

In the following, we introduce several meta-models that support source code, history and change representation or reason about the evolution of a software system.

FAMIX. FAMIX [Ducasse 2000, Tichelaar 2000] is a family of meta-models for representing the source code of object-oriented systems. FAMIX offers a language-independent, first-class representation of object-oriented, class-based languages that has been used by a wide range of software engineering tools such as the Moose[5] platform for software and data analysis.

EMF – ECore and GenModel. ECore[6] is a source code meta-model for describing Java models and the run-time support for the models within the Eclipse Modeling Framework (EMF). EMF instantiates models that conform to an ECore meta-model by code generation. The Java classes are generated from an ECore meta-model specification. This process is performed in two steps: (1) the ECore meta-model is transformed into a GenModel[7] model that can contain additional implementation-specific information. (2) a model-to-text (M2T) transformation consumes the GenModel in order to generate the functional Java code.

Hismo. Gîrba [Gîrba 2005a, Gîrba 2006] proposed Hismo, a history-based approach that extends the FAMIX source core meta-model to provide facilities for representing and reasoning over the history of a software system. It represents multiple versions of a system within a single *history* model. A version is a snapshot of the system, a complete FAMIX representation of that version along with information that relates source code entities over various versions.

SpyWare. Robbes and Lanza [Robbes 2007, Robbes 2008b] developed the SpyWare change-aware development toolset. It tracks and reasons about the changes a developer makes to a program within the integrated development environment (IDE). SpyWare provides a fine-grained model that represents changes as first-class entities. With this change-based evolution model, systems are represented as evolving abstract syntax trees (ASTs), composed of domain-entities (packages, classes, methods, variables and statements) specific for the base language (Smalltalk and Java). Each entity (AST node) has a change history containing all the changes representing one particular state of the program during the system's evolution.

ChEOPS. Ebraert [Ebraert 2007] presented ChEOPS, a proof-of-concept implementation of change-based feature-oriented programming (FOP). The underlying meta-model is an extension of the FAMIX source core meta-model that represents fine-grained first-class change objects based on development actions. ChEOPS shares the objectives and advantages with SpyWare. However, ChEOPS does not represent a program as evolving abstract syntax trees; instead it allows developers to define

[5]Moose: http://www.moosetechnology.org

[6]ECore: http://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/ecore/package-summary.html

[7]GenModel: http://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/codegen/ecore/genmodel/package-summary.html

their own domain-specific and high-level changes, apply those changes, undo changes and verify the preconditions of the changes to ensure the application consistency.

Syde. Hattori and Lanza [Hattori 2009a, Hattori 2009b, Hattori 2010] proposed Syde, a client-server toolset for synchronous development of Java systems. Syde extends Spyware's change-based evolution model to deal with changes made by multiple developers working in parallel. The evolution of a system is modeled as a set of sequences of changes, where each sequence is produced by one developer. On the client-side, Syde traces two types of change operations (atomic changes and rename/move refactorings). On the server-side, it keeps one AST per developer, which reflects the state of the system at a developer's workspace.

Orion. Laval [Laval 2009, Laval 2011] developed Orion, an interactive prototyping tool for software reengineering that allows developers to simulate changes and compare their impact on multiple versions of software source code models. Orion's meta-model is an extension of the FAMIX source code meta-model designed to optimize memory usage of multiple versions for large models. Orion's meta-model supports the sharing of entities between versions. To save memory space and decrease creation time, program entities which do not change are shared between different versions of the model.

Other approaches

Ebraert and Molderez [Ebraert 2010] extended the change model underlying ChEOPS [Ebraert 2007] with the notion of intensional changes – descriptive changes that can evaluate to an extension of changes – to support the modularization of crosscutting features. Meyers *et al.* [Meyers 2010] use that work on intensional changes to avoid co-evolution in the domain of change-based feature-oriented programming. Ebraert *et al.* [Ebraert 2011] makes use of the ChEOPS' meta-model to present a bottom-up approach for generating Feature-Oriented Domain Analysis (FODA) diagrams from the changes to the source code, thus bridging the gap between feature-oriented design and implementation.

Orthogonally to the mentioned modeling approaches, there exist a number of language representation toolkits such as CDT[8] (Eclipse's C/C++ Development Tooling) or Necula *et al.*'s [Necula 2002] approach CIL, a high-level source code representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs. These approaches do not address software history.

Discussion

The use of first-class change objects has been applied in a variety of fields. Syde uses first-class changes to increase awareness of changes made by other developers in multi-developer projects. SpyWare has shown that the fine-grained changes contain a lot more information about the evolution of a software system than when using snapshot-based information. ChEOPS has been used in the context of FOP combining changes into features [Ebraert 2008]. Orion has been used to represent the future by simulating scenarios of multiple versions of a system in the context of software reengineering.

[8]CDT: www.eclipse.org/cdt

While existing meta-models provide a rich medium to model source code or changes, none of the existing approaches provide a complete unified representation of a system's source code, history and individual changes, and reason over the history and changes in the context of branching and merging. Furthermore, the Hismo meta-model does not scale for large histories, and it is unclear how existing models for representing changes scale with respect to large numbers of changes.

3.5.2 Towards Conflict Resolution and Merging

Version control systems (CVS, Subversion, Git, Mercurial, ...) allow developers to merge changes. These systems are widely used in collaborative development, and thus merging techniques, and conflict detection and resolution have been widely studied [Mens 2002].

Merging techniques can be categorized depending on the algorithm they use (two-way or three-way), on how they represent the software artifacts (text, trees, graphs), and on which information they use during the merge process.

Integrating changes from one branch into another, for which they were not originally intended is known as *cherry picking*. Changes from the source branch may be incompatible with the target branch in various ways. Code within or near the changed region evolved within the two branches in different ways, making the affected code fragments syntactically different. More subtle, the semantic structure of the two branches can be different, *e.g.,* in terms of variable bindings or inheritance hierarchies. Detecting and resolving these incompatibilities is known as *conflict resolution*.

Merging algorithms. The *two-way algorithm* attempts to merge two versions of a software artifact without relying on the common ancestor from which both versions originated [Hunt 1976,Hunt 1977]. The *three-way algorithm* uses the information of the common ancestor during the merge process, allowing the detection of more conflicts [Lindhom 2001,Khanna 2007]. As a result, three-way merging is more widely used by merge tools than two-way merging.

Text-based merging. Text-based merge approaches [Leblang 1984, Tichy 1985, Adams 1986, Berliner 1990,Lubkin 1991] consider software artifacts as *text* (or binary) files (*i.e.,* ignoring semantic information). Commonly they use *line-based* merging, where lines of text are taken as indivisible units [Hunt 1976]. Line-based merging detects parallel modifications (inserted, deleted, modified, or moved) of text lines, but because this level is too coarse-grained, it cannot combine well two parallel modifications to the same line. In spite of this, this approach remains a very useful technique because of its efficiency, scalability, and accuracy. CVS, Subversion and other open-source or commercial configuration management tools use text-based merging.

Syntactic-based merging. Syntactic-based merging [Buffenbarger 1995] is more powerful than *textual merging* because it takes the syntax of software artifacts into account. This technique ignores conflicts related to code comments, line breaks or tabs. Different categories of syntactic merge systems exist depending on the data structure they manipulate, parse-trees or abstract syntax trees [Grass 1992,Asklund 1994,Yang 1994] and graphs [Mens 2000,Rho 1998]. A disadvantage of this technique is that it is unable to detect some conflicts when the merged program is syntactically correct but semantically incorrect.

Semantic-based merging. Semantic-based merging improves syntactical merging as it takes the *semantics* of the changes into account. This kind of merging deals with (static) semantic conflicts and behavioral conflicts. The context-sensitive merge approach of Westfechtel [Westfechtel 1991] detects static semantic conflicts by augmenting the AST with relationships that express the bindings of identities to their declarations. More advanced semantic-based approaches [Horwitz 1989, Yang 1992, Jackson 1994, Berzins 1994, Binkley 1995] detect behavioral conflicts using denotational semantics, program dependence graphs, and program slicing.

Structural-based merging. Structural-based merging uses the structure and semantics of the software artifacts to resolve merge conflicts automatically. This kind of merging is oriented to solve the conflict-resolution problems of text-based and syntactic-based merging. Furthermore, despite of this common definition two interpretations exist about structural-based merging.

Based on the first interpretation, structural merging is language dependent. It operates on AST or similar representations, rather than plain text, and adds extra information on the underlying language it supports. By means of this, structural merging decreases general applicability but increases expressiveness to handle more merge conflicts automatically. Apel *et al.* [Apel 2011] mix *structural-based* and *text-based* merging to propose a semistructured merge language-dependent approach where structural information about the artifacts is declaratively added in the form of annotated grammars to assist automatic resolution of certain semantic conflicts (*e.g.,* related to renaming).

Based on the second interpretation, structural merging is oriented to deal with refactorings or restructuring transformations that are behavior preserving (*i.e.,* they affect the structure of an artifact but not its semantics). Then structural merge conflicts arise when one of the changes to an artifact is a restructuring and the merge algorithm cannot decide in which way the merged result should be structured. Hattori and Lanza [Hattori 2009b, Hattori 2010] mix *change-based* and *structural-based* merging in Syde. Atomic operations and rename/move refactorings are recorded in a multi-developer environment. Syde keeps the changes of each developer in a separate AST, and detects emerging structural conflicts by comparing a developer's AST with the others' ASTs after a new atomic operation has been applied.

State-based Merging. State-based merging considers only the information in the original version and/or its revisions during the merge. A two-way algorithm is used in state-based merging.

Change-based merging. Change-based merging uses additional information about the precise changes that were performed during evolution of the software (*e.g.,* the intermediate changes between committed files that are not kept in source repositories). This additional information can support the detection of semantic conflicts.

Operation-based merging. Operation-based merging is a kind of *change-based* merging that models changes as explicit operations or transformations [Feather 1989, Lippe 1992, Berlage 1993, Mens 2000, Shen 2004]. These sequences of change operations, referred as command histories [Berlage 1993], correspond to the commands issued to the integrated development environment (IDE).

This approach can improve conflict detection and allows better conflict solving [Feather 1989, Lie 1989, Lippe 1992, Mens 1999]. Edwards' operation-based framework detects and resolves semantic conflicts from application-supplied semantics of operations [Edwards 1997]. GINA [Berlage 1993] merges command histories using a redo mechanism to apply one developer's changes to other's version. This approach cannot handle well long command histories and fine granularity. MolhadoRef [Dig 2008] mixes operation-based and state-based merging to capture the semantics of refactoring operations. It records the refactorings performed by the developers, and calculates deltas that represent other changes. Therefore, it can merge changes that involve a combination of logged refactorings and textual editing.

Delta algorithms. Delta or difference algorithms are used to calculate the difference (or delta) between a version and one of its revisions. They can be categorized according the types of deltas that can be distinguished:

- *Symmetric vs. Directed deltas.* A *symmetric delta* calculates the difference between two versions V_1 and V_2 as the set of difference $V_1 \backslash V_2$ or $V_2 \backslash V_1$. A *directed delta* specifies the difference between two versions as a sequence of modification operations. Symmetric deltas are typically used in the context of two-way merging, whereas directed deltas are useful in the context of three-way operation-based merging.

- *Textual, Syntactic, or Semantic deltas.* They are related to the merging approach in which they are used, that is for text-based, syntactic-based and semantic-based merging. *Textual deltas* are obtained by comparing two text files, finding the longest common substring and then computing a distance delta from this common substring [Hunt 1977]. The Unix *diff* utility [Hunt 1976], the *bdiff* [Tichy 1984] and *vdelta* algorithms process textual deltas. *Syntactic deltas* are the result of comparing two syntax representations (*e.g.,* parse trees). Yang [Yang 1991] describes a comparison tool for detecting syntactic differences between programs. *Semantic deltas* contain the semantic differences between two versions of a program. *Semantic Diff* [Jackson 1994] expresses a semantic delta in terms of the observable input-output behavior.

- *Forward or Backward deltas.* They are established based on how subsequent versions of a software artifact are stored. With *backward deltas* [Rochkind 1975], the latest version is stored entirely and previous versions are stored as deltas. With *forward deltas* [Tichy 1985], the originally version is stored entirely, while newer versions are expressed as deltas to the original one. Backward deltas are widely used even though they are less intuitive. They speed up retrieval of the last and probably most frequently accessed revision.

- *State-based or Change-based deltas. State-based* version control systems [Rochkind 1975, Tichy 1985] calculate the difference between a revision and its ancestor version, and store only this difference instead of the entire revision. *Change-based* approaches are identified as intensional or extensional. *Extensional* (or embedded deltas) versioning [Asklund 1994, Rho 1998] processes annotated changes for each version. *Intensional* (or change set model) versioning [Gulla 1991] processes changes that can be specified independently from the versions to which they are applied. Operation-based merging relies on an intensional change-based model

Other approaches

Shao *et al.* [Shao 2007] show that there is a linear correlation between the degree of parallelism and the likelihood of a defect in the changes. However, textual analysis can only detect a very small portion of change interferences. To detect change interference at the semantic level, Shao *et al.* subsequently implemented a tool, SCA, that combines data dependency analysis and program slicing [Shao 2009].

Darcs[9] is a distributed change-based source-code management system based on an algebra of patches, named *the theory of patches*, for manipulating changes. This theory is about the commutation, or reordering, of changes in such a way that their meaning does not change. The Darcs merge operation is based on the patch commutation algorithm. Darcs supports – similar to the Git version control system – *cherry picking* allowing users to choose the patches that they want to check in or check out. However, semi-automatic handling of conflicts and merging of features are not well supported.

Discussion

Existing semantic merging techniques tend to be complex and require manual annotation of the source code with meta-data to guide the merging process. Most of the existing merge tools and techniques provide no support for detecting structural merge conflicts as they can not infer the needed information from the source code only.

Merging techniques used by popular VCS (*e.g.,* CVS, Subversion, Git) are based on simple, text-based algorithms, and are therefore oblivious to the program entities they merge. Even though there exist other approaches providing advanced merging support [Apel 2011] that significantly reduce the amount of merging conflicts, such approaches do not support integrators in identifying redundant changes or changes that introduce inconsistencies at the level of the design of the target system.

These approaches do not provide analyses to understand the dependencies between changes. The integrators are left to manually compare changes within the input stream of changes, and assess how these changes may impact the target system. Such work is particularly tedious between product forks, where the distance between branches grows larger over time.

3.5.3 Change Impact Analysis

The integration of a change into a system requires prior assessment of the potential impact of that change on the system's behavior. The behavior can be greatly impacted when a change made for a branch is merged with another branch in the case that both branches differ in purpose or design. Here this assessment is required.

Impact analysis is among the major issues related to software change management. Understanding the impact of changes has been considered in areas including software maintenance, program refactoring [Mens 2004] and test prioritization [Elbaum 2000].

Program slicing approaches. Program slicing [Weiser 1981] provides an in-depth analysis of impact of the changed code. Several approaches, such as CodeSurfer [Anderson 2001], use pro-

[9]Darcs: http://darcs.net

gram slicing to understand which part of a program is impacted by a variable, and which are the parts of a program that can impact a particular variable or a given source-code element in general [Bohner 1996, Tip 1995, Welser 1984]. In the context of software maintenance, such program slicing techniques have been widely adopted [Gallagher 1991].

Conceptual Framework. Ryder and Tip [Ryder 2001] proposed a conceptual change impact analysis for object-oriented programs in terms of affected regression or unit tests [Elbaum 2003]. The authors introduce techniques that could determine the tests that are affected by a set of changes, and the subset of changes responsible for the failure of each affected test. They propose to perform the change analysis on (coarse-grained) atomic changes extracted from source code edits (*e.g.,* added empty class, added empty method) and their syntactic dependencies by ordering atomic changes (*e.g.,* a method m can be added to class X if that class exists).

Chianti. Ren *et al.* [Ren 2004] proposed Chianti as the implementation and extension of the proposal of Ryder and Tip [Ryder 2001]. Chianti is a (semantic) change impact analysis tool that decomposes the difference between two versions of a Java program into a set of extended atomic changes (*e.g.,* changed definition of an instance initializer) and their interdependences. They use a pairwise comparison of the abstract syntax trees of the classes in both versions. Change impact analysis is performed on the (dynamic) call graphs derived from a set of regression or unit tests applied to both versions. Chianti determines the affected tests whose behavior have been modified by the applied changes, and the affecting changes for each affected test. Chianti is one of a large category of approaches related to test prioritization [Elbaum 2000], and serves as the foundation for several approaches that we explain below.

Crisp. Chesley *et al.* [Chesley 2005] presented Crisp, a tool that assists developers in isolating relevant subsets of changes that directly cause the failure of a regression test. Crisp leverages and augments Chianti [Ryder 2001] to detect failure-inducing changes between two versions of a Java program. It allows developers to build compilable intermediate versions of a program with partial changes that can be applied to the original version to ensure compilation in order to locate the exact reason for the failure. This approach was improved in [Ren 2006] where the original dependence relationships were refined. Three kinds of dependences between atomic changes that capture syntactic and partially semantic dependences: (a) *structural dependences* capture the necessary sequences that occur when new elements are added or deleted in a program; (b) *declaration dependences* capture all the necessary element declarations that are required to create a valid intermediate version; and (c) *mapping dependences* are implicit dependences introduced by atomic changes such as overloading methods that may affect the behavior of a method despite the fact that no textual changes occur within that method.

JUnit/CIA. Stoerzer *et al.* [Stoerzer 2006] proposed a change classification tool, JUnit/CIA that helps programmers to find failure-inducing changes (between two versions of a Java program) according to the tests that the changes affect. They rely on the change impact analysis tool Chianti [Ryder 2001] to extract atomic changes, affected tests and affecting changes. Then, they classify changes as Red (changes are highly likely to be the reason for the test failures), Yellow (changes are possible problematic), or Green (changes are correlated with successful tests) according to five classifiers. These classifiers are based on the JUnit test result model (pass, fail, crash). For coverage issue, their change classification techniques also classify changes that do not affect any test as Grey.

Celadon. Zhang *et al.* [Zhang 2008] proposed the tool Celadon, a change impact analysis framework that uses an atomic change representation to capture the semantic differences between two versions of an AspectJ program. They built a change impact model based on static AspectJ call graphs to determine the affected program fragments, affected tests and their responsible affecting changes. The change impact model relies on the computation of atomic changes and their inter-dependence relationship. The authors defined a catalogue of 21 types of atomic changes (*e.g.,* changed pointcut body) for AspectJ programs. They extended the concept of atomic changes proposed in [Ryder 2001] to aspects.

Safe-commits. Wloka *et al.* [Wloka 2009] presented Safe-commits, an analysis-based algorithm to identify committable changes that can be submitted early, without causing failures to existing tests in the repository, even in cases when failing tests exist in a developers' local code base. The idea is to decrease the time interval between commits, by establishing three commit policies (Restrictive, Moderate, and Permissive) that depend on the test result model and enable developers to release their changes often. This approach relies on the data generated by Chianti [Ren 2004]. Safe-commits takes into account all atomic changes, the affected tests (according to a commit policy), the exercised changes by each test (changes used by a test), and the covered changes by each test (exercised changes by a test that are applied to their dependencies as well). The output of the algorithm is a set of changes that do not break existing tests and can be committed.

Reuse Contracts. Lucas and Steyaert [Lucas 1997, Steyaert 1996] introduced reuse contracts as an object-oriented methodology to assist software engineers in understanding how a component can be reused, adapting components to particular needs, and estimating the impact of changes. With reuse contracts a component is reused on the basis of an explicit contract between the provider of the component and a reuser that modifies this component. The provider documents how the component can be reused, and the reuser documents how the component is reused or how the component evolves. Their contract clauses allow to detect what the impact of changes is, and what actions the reuser must undertake to upgrade if a certain component has evolved. Reuse contracts help in keeping the model of the provider consistent with the model of the reuser. They were used at the implementation level to express reuse in evolvable class inheritance hierarchies [Steyaert 1996], and reuse and evolution of collaborating classes [Lucas 1997].

Other approaches

Kung [Kung 1994] categorized various types of changes of object-oriented systems and a formal model for capturing and inferring the impact of class library changes to identify affected components. Several techniques aim at identifying the so-called fragile bass class problem [Mikhajlov 1998], that arises when changes in a framework have unexpected impacts on framework instantiations. For example, changing the visibility of a method from protected to private may break classes that extend the framework.

Han [Han 1997] proposed an approach to support impact analysis and change propagation in software engineering environments. This approach is focussed on how the system reacts to a change. They use the environment representation of artifacts (variables, method, classes) and their dependencies (association, aggregation, inheritance, and invocation) that can be impacted. Later, Abdi *et al.* [Abdi 2006] reused this work for their change impact analysis. Chaumun *et al.* [Chaumun 2002] defined a class-based change impact model. Their approach is similar to the previous ones but the model is more complete and systematic. The impact of a change is calculated to ensure that the system will still run correctly after the change is implemented.

Alam *et al.* [Alam 2009] used change dependency graphs [German 2009] to examine how changes build on each other over time and determine the impact of these changes on the quality of a project. The authors showed that time dependences vary across projects and throughout the lifetime of each project. They also found that changes built on top of new code (instead on stable code) are more defect prone.

Herzig [Herzig 2010] introduced the concept of *transaction dependency graph* based on the notion of change genealogies defined by Brudaru and Zeller [Brudaru 2008]. This is a similar approach to the change impact graphs by German *et al.* [German 2009], but differs considering version control transactions instead of atomic changes to define multiple dependency metrics on these change genealogy graphs.

Law and Rothermel [Law 2003] defined a new technique – PathImpact – for impact analysis based on dynamic information obtained through simple program instrumentation. They execute a program with a set of inputs, collecting compressed traces for those inputs, and using the traces to predict impact sets. PathImpact does not rely on availability of program source code and does not require static dependency calculations.

Badri *et al.* [Badri 2005] propose a change impact analysis based on a call graph for making impact predictions. This technique restricts the scope of the analysis by only considering methods within the reachable paths of the call graph. Abdi *et al.* [Abdi 2009] propose a technique based on a probabilistic model, where a Bayesian network is used to analyze the impact of a given scenario.

Discussion

Several approaches based on the conceptual framework presented by Ryder *et al.* [Ryder 2001] and extending Chianti [Ren 2004] have proposed change impact analysis using an atomic representation of changes and the notion of dependencies between these changes. They have been used to identify failure-inducing changes, to capture the semantic differences on the context of aspect programming, and to identify changes that can be safely committed to decrease the interval of time between commits. They makes use of a change representation and dependencies which we intend to do. However, they can only compare two versions of a program. No history and streams of changes is supported in the presence of branches.

Reuse Contracts have been proposed for understanding how components can be reused, adapting components and estimating the impact of changes. Other approaches have been applied in several contexts: to infer the impact of change library changes, to calculate the impact of changes and support change propagation, to examine how changes build on each other and determine the impact in terms of quality, to perform dynamic impact analysis, and to predict the impact of changing methods using call graphs.

While there exists an extensive and varied work on change impact analysis, none of these approaches is dedicated to assess the impact of changes as a means to support integrators deciding which changes should be integrated. Especially to perform cherry picking of changes between multiple branches that may have substantially evolved apart. These approaches are complementary to our work and form a good foundation for providing change impact analysis to assist the integration process.

3.5.4 Change Dependencies

One of the cornerstones of our approach, as will be explained in Chapter 6, is change dependencies. We are not the first making use of the dependencies between changes. However, within literature, there are not many related approaches using dependencies. The existing approaches are limited to support change impact analysis.

GENEVA. Herzig and Zeller [Herzig 2011] proposed GENEVA, an approach that extracts change dependency graphs (known as change genealogy) to use them as a model to analyze change impact. By means of static analysis, they establish a change genealogy that order changes by dependencies (a change CH_2 depends on a change CH_1 if CH_2 uses or modifies code defined or previously changed within CH_1). They determine dependencies between changes across transactions (files that changed together) in version archives. Later, they apply model checking to the change genealogy to determine frequent rules expressed in computation tree logic (CTL) that establish temporal process patterns. These patterns encode key features of the software process such as pending development activities, and serve to provide recommendations for developers. As an example usage, the authors use GENEVA to predict long-term change couplings – change couplings spanning multiple revisions – with a precision of 70%.

Change Impact Graphs (CIGs). German *et al.* [German 2009] presented a method that determines the impact of historical code changes on a particular code segment. Their approach guides developers to investigate failures in unchanged functions that are affected by bugs introduced in prior code changes. First, they extract dependence graphs from C programs stored in version control systems that represent the evolution of a system (with the changes tracked at line level). Second, they generate CGIs from the dependence graphs, a reported location of a failure and the period of interest in the history to look for prior changes (both provided by a developer). Third, they prune and annotate the CGIs if the developer specifies areas of interest and annotation rules. The developers investigate the CGIs to pinpoint the changes which mostly likely introduced the bugs, causing the reported failure. Note that this approach is applied to procedural code and the dependence graphs take into account changes (*i.e.,* additions, modifications and deletions) of function calls but neither function pointers nor polymorphic function calls.

Change Genealogies. Brudaru and Zeller [Brudaru 2008] introduced change genealogies as part of their ideas to assess the long-term impact of changes by measuring the impact in terms of quality, effort, and maintainability. They propose to express the history of a system with change genealogy graphs (*i.e.,* directed acyclic graph of changes) that contain the sequences of changes which take place in a system, as extracted from version archives, and incorporate dependencies between changes, derived from the changed code. Having $CH_1 \rightarrow CH_2$ means that the change CH_1 enables (and thus impacts) a change CH_2; the change CH_2 thus depends on CH_1 (*e.g.,* CH_1 defines the readline() function and CH_2 uses the readline() function). Dependencies could be established iff: (a) change CH_1 has an earlier timestamp than change CH_2; (b) a set of common identifiers appear in both changes, and (c) applying CH_2 to a version of the system but not CH_1 would not compile.

Chianti and extensions. In Section 3.5.3, we introduced Chianti [Ryder 2001] as the implementation of the change impact analysis proposed in [Ryder 2001]. This approach, and the others derived from it [Chesley 2005, Ren 2006, Stoerzer 2006, Wloka 2009] are also situated in this category of related work as they rely on the computation of dependencies between atomic changes for identifying which changes affect the behavior of tests. Change dependences were summarized as syntactic dependences, that is, an atomic change CH_1 is dependent on another atomic change CH_2, if applying CH_1 to the original version of the program without also applying CH_2 results in a syntactically invalid program (*i.e.,* CH_2 is a prerequisite for CH_1). Where syntactic dependencies do not capture all semantic dependences between changes (*e.g.,* consider changes related to a variable definition and a variable use in two different methods). Later in [Ren 2006] dependences were refined and classified in three categories: structural dependences (*e.g.,* changing definitions of a field or method), declaration dependences (*e.g.,* abstract method declaration and implementation), and mapping dependences (*e.g.,* overloading methods). Note, that their dependences are based on the declaration of atomic changes for Java programs.

Discussion

Dependencies between changes have been successfully used in several impact analyses. GENEVA has used dependencies to find patterns that represent features of the software process (*e.g.,* long-term change coupling). CGI has built dependencies graph to determine the impact of prior changes on failing unchanged code. Change Genealogies have been introduced to assess the long-term impact of changes in terms of quality, effort, and maintainability. Chianti and the approaches derived from it have been focused on detecting failure-inducing changes between two versions.

While these approaches make use of dependencies between changes to support impact analysis, none of them is focused on supporting assessing of changes for the integration process. These analyses do not support activities such as cherry picking. This is needed when integrators have to move changes back and forth between branches. Here the dependencies of changes play an important role for the integration of complete changes. Providing impact analysis for aiding integrators will definitely enhance our approach.

3.5.5 Understanding Development Tasks

Several approaches exist that use questions as a means to understand developments tasks (*e.g.,* maintenance or code comprehension) and to identify the developers' information needs. By understanding such information it is possible to provide adequate support and by means of the questions is possible to assess such support. We followed the same approach as explained in previous sections of this chapter. Here we present an extended description of relevant studies.

Questions about Code. LaToza and Myers [LaToza 2010] conducted a survey to investigate the real questions that developers ask and experience problems answering. Part of their survey included a free response listing questions. From the answers of 179 developers at Microsoft to that part of the survey they obtained 371 *hard-to-answer* questions (*e.g.,* "Are the benefits of this refactoring worth the time investment?", "Is this functionality already implemented?" or "How does this code interact with libraries?"). These questions were divided in 10 categories using the underlying intent, such as rationale, debugging, policies, history, implications, implementing, refactorings, teammates, building and branching, and testing. They concluded that having a better understanding of developers' information needs may lead to new tools, programming languages, and process that make *hard-to-answer* questions less time consuming or error prone to answer.

Study on Integration Decisions. Phillips *et al.* [Phillips 2012] performed an study focused on how developers of a large-scale system make branching and integration decisions while managing releases. Semi-structured interviews were conducted with seven professionals of the same company using the branching and merging survey proposed in [Phillips 2011]. The authors found that developers making decisions need to consider 10 factors including code churn (line of code changed and not yet integrated), potential conflicts, bugs counts, and dependencies between branches. The authors also identified the information needed to support integration decision-making in a branched context (parallel development). Release decision makers need to predict storms of conflicts, detect pressure building up from non-integrated changes (the code churn that has built up in a branch), monitor code flow between branches (how frequently integrations are occurring between branches), and track branch health (metrics such as test results, bugs, and task completion at branch level).

Empirical Study on Branching and Merging. Premraj *et al.* [Premraj 2011] presented an empirical study that observed developers branching without considering the consequences on merging. The goal of the study was to understand the implications of such branching for the cost of merging changes. The study had two parts: 1) A qualitative study where 16 software professionals were surveyed (5 questions oriented to branchers and 3 questions oriented to mergers) to learn their views on branching and merging files, and their experience with the development overhead from branching and merging. 2) A quantitative study that calculated the number of branches, the number of merges on a number of files, and the time spent on merging files. From the study they established (a) the roles of the branchers and mergers (*i.e.,* architects, configuration managers, integrators and developers), and (b) the types of files that dictate the cost of merging (*e.g.,* configuration files). They concluded that software configuration management (SCM) tools and SCM best practices (*e.g., branch only when necessary, branch late, propagating early and often*) are not sufficient to share files in an agile development environment. They also suggested that contents of shared files must be aligned with the responsibilities of the primary owners of those files, as a way to decrease conflicts of branching and merging files.

Questions related to Evolution Tasks. Sillito *et al.* [Sillito 2008] proposed a catalogue of 44 types of questions programmers ask during software evolution tasks. The authors' goals were to better understand what a programmer needs to know about a code base when performing a change task, how a programmer goes about finding that information, and how well today's programming tools support answering their questions. They performed two qualitative studies [Sillito 2005, Sillito 2006] observing 9 and 16 programmers respectively, making changes to medium and large sized code based. From the analysis of the empirical information collected during both studies, they established the used tools, type of change tasks, paired versus individual programming, and the level of prior knowledge of the code base. 44 questions were classified in 4 categories: (a) finding focus points (*e.g.,* "Where in the code is the text in this error message or UI element?"), (b) expanding finding points (*e.g.,* "Where is this method called or type referenced?"), (c) understanding a subgraph (*e.g.,* "How are instances of these types created and assembled?"), and (d) questions over groups of subgraphs (*e.g.,* "What will the total impact of this change be?"). They also established that 34% of the questions was fully addressed by tools and 66% of the questions only partially addressed. From the results, they found that programmers need better tool support for asking more refined or precise questions, maintaining context, and piecing information together.

Information Fragment Model. Fritz and Murphy [Fritz 2010] presented a study in which they interviewed 11 professional developers to identify different kinds of questions they need answered during development, but for which support is lacking. From the results, they established a catalogue of 78 questions classified in several categories such as people specific (12 questions *e.g.,* "Which code reviews have been assigned to which person?"), change code specific (35 questions *e.g.,* "What are the changes on newly resolved work items related to me?"), work item progress (11 questions *e.g.,* "Which features and functions have been changing?"), and so on. Alongside this study, they introduced the information fragment model (*i.e.,* a subset of development information for the system of interest) and associated prototype tool built on top of Eclipse[10] for answering the identified questions by composing different kinds of information needed. This model provides a representation that correlates various software artifacts (source code, work items, team membership, comments, bug reports, and others). By browsing the model, developers can find answers to particular development questions.

Discussion

The two studies related to branching and merging were performed by asking concrete questions to developers about these activities. The first was focused on understanding the implications on branching on the cost of merging changes [Premraj 2011]. The second was focused on supporting integration decisions [Phillips 2012]. The other three approaches gathered the questions by observing developers performing their work or by directly getting the questions from the developers. These approaches established catalogues of questions that are intended to identify the information needs of maintenance tasks.

These approaches have proven the importance of gathering developers' questions or their needed information regarding development tasks to better understand the complexity of such tasks and to be

[10]Eclipse: www.eclipse.org

able to provide adequate support. Even though, these studies propose large set of questions, many of these questions are not specific enough or do not apply to the context of integrating streams of changes. We complemented our results with relevant questions from these studies.

3.5.6 Other Related Work

Logic Program Querying and Meta-programming Languages

Program query languages [Wuyts 2001, Janzen 2003, Hajiyev 2006, Hou 2006, De Volder 2006] allow writing custom queries that extract information from the source code of a system. SOUL [Wuyts 2001] is a logic-based program querying language to reason over the structure of object-oriented systems. While the SOUL language is very similar to PROLOG, it provides a number of specialized features (such as linguistic symbiosis) that facilitate reasoning over software systems, as well a set of logic libraries that offer dedicated predicates for reasoning about programs written in Smalltalk, Java, C(++) and Cobol.

The JQuery tool [De Volder 2006] uses a PROLOG dialect to offer an expressive means to query source-code entities and the relationships between these entities. The work of Verbaere and De Moor concerning CodeQuest [Hajiyev 2006] and SemmleCode [de Moor 2007] provides a different approach that favours performance over expressivity. These approaches use respectively Datalog [Ceri 1989] and QL, languages that only offer a subset of the PROLOG language by *e.g.,* limiting the possible forms of recursion and excluding the definition of data structures.

Meta-programming languages allow developers to write programs that generate, analyze or transform other programs. Rascal[11] is a new meta-programming language for source code analysis and manipulation. Rascal programs can read, analyze, transform, generate and/or visualize other programs. It has been designed following the Extract-Analyze-SYnthesize (EASY) paradigm [Klint 2011]. Rascal can be applied to several domains such as compiler construction, implementing domain-specific languages, constraint solving, software renovation and so on.

Querying Source Code History

Kellens *et al.* [Kellens 2011] propose ABSINTHE, a logic-based program query language that supports querying versioned software systems using logic queries. It extends the SOUL program query language with quantified regular path expressions for reasoning about a system's history. These quantified regular path expressions exhibit the properties of each individual version in a sequence of successive software versions.

In previous work we have proposed *Time warp* [Uquillas-Gómez 2009], a prototype implementation that extends the SOUL program query language to allows developers to write queries about the history of a system. It is based on the FAMIX and Hismo meta-models and offers an ad-hoc specification language (library of dedicated predicates) to reason about these models, as well as to express temporal relationships between the entities in both models.

Hindle and German [Hindle 2005] propose SCQL, a dedicated formal model and query language for reasoning over source code repositories. A repository is instantiated as a formal model to serve as the underlying model which they reason about. The model is a graph in which the different entities (*e.g.,* revisions, files, authors) stored in the repository are vertices and their relationships are edges.

[11]Rascal: http://www.rascal-mpl.org

SCQL supports temporal logic operators such as *previous*, *after*, *always*, *never*, etc. used to express queries.

Source Code Change Extraction
Fluri *et al.* [Fluri 2007] propose *change distilling*, a tree differencing algorithm for fine-grained source code extraction. They identify changes between two Java programs by finding both a match between the nodes of the compared two abstract syntax trees and a minimum edit script that can transform one tree into the other given the computed matching. The authors improved the existing tree differencing algorithm by Chawathe *et al.* [Chawathe 1996] to classify change types based on a taxonomy of source code changes that Fluri and Gall established in a prior work [Fluri 2006]. This taxonomy defines changes according to tree edit operations (insert, delete, move, update) in the AST and classifies each change type with a significance level (*e.g.,* else-part insert (high), attribute renaming (high)).

Mining Software Repositories (MSR)
MSR refers to the extraction and processing of information stored by version control systems, such as SVN, CVS, Git. Hassan proposes [Hassan 2009] a technique to predict faults in a system by applying complexity metrics on the changes that are present in the repository. Source Sticky Notes [Hassan 2004] is an approach that annotates a static dependency graph of a system with information that is extracted from the history of a system, to help developers to understand the context of the changes they are applying. DynaMine [Livshits 2005] is a tool that applies data mining techniques on version archives to find common usage patterns by analyzing co-changed methods.

Software Classifications
De Hondt [Hondt 1998] proposes *software classifications* as a means to recover architectural elements in evolving object-oriented systems. This approach is based on Reuse Contracts that we described before. Software classifications consist of a model and a technique. The software classification model provides simple concepts to organize large software systems and their evolution in manageable units (classifications). The software classification technique provides strategies to set up and recover those manageable units. Software classifications have been applied to: (a) expressing multiple views on software, (b) recovering of collaboration contracts, (c) recovering of reuse contracts, (d) recovering of architectural components, and (e) management of changes.

Aspect-Oriented Software Analysis
Within the field of aspect-oriented software development, numerous techniques have been proposed that mine for crosscutting concerns [Kellens 2007], such as the work of Marin [Marin 2007], who uses fan-in analysis to identify possible aspect candidates.

3.6 Conclusion

In this chapter we have presented the intent of our solution: a comprehensive tool suite to provide integrators access to the information discussed in Section 3.4.

We covered four topics related to changes that complement the problems inherent of the integration process in a collaborative environment (see Chapter 2). The use of branching and merging make the integration process a very complex task. Unfortunately, current tools do not provide the adequate support for developers who integrate changes within a single branch or between branches. Developers lack tools that aid them in understanding the context of changes in an efficient manner. Such activity is mostly manual and time consuming. Merging changes between branches is even more complicated, and no tools exist to identify and understand changes that can be applied from one branch to another, that is supporting cherry picking.

An overview of the integration process as it can be found in open-source development was briefly described as a background to introduce the definitions and terminology that we use in this dissertation. Even though, most of the terminology is well-known (*e.g.,*, commits or history), we provided our definitions in the context of the integration of changes. We introduced terms such as *stream of changes* and *delta dependencies* to refer to a sequence of set of changes within a branch, and to the dependencies between these sets of changes, respectively. Note also that some definitions are tailored to support integration in Pharo, but they can be refined and applied to other infrastructures.

The second topic presented in this chapter was a catalogue of 64 questions that developers ask when they are integrating or want to integrate changes. We conducted a study to gather such questions as a means to identify and understand the developers' information needs that can ease the answering of the questions, and therefore to support them integrating changes. Moreover, these questions serve as the foundation to assess our contributions in Chapters 5 and 7. We described the methodology used for our study, the data and results obtained. However, as we do not only intend to base our contributions on our identified questions, but on to relevant questions that integrators raise independently of the programing language or tools used, our findings were also extended and verified with other questions found in similar but broader studies. The 64 questions were clustered in 5 categories, and for each category a description was added. This part ended with a discussion about how tools support or can support answering of these questions.

The third topic described the information that can be used to define the requirements for our solution presented in Chapter 2.4. Based on the questions, we identified 12 kinds of information such as size, author, time, structure, change scope, vocabulary, dependencies, and so on that we can use for the change characterizations. Additionally, we presented a summary including which questions can be answered by each kind of information, and to which extent the support can be provided.

The last topic presented the state-of-the-art of relevant work for the context of this dissertation. We focused on five topics: (a) modeling source code, history and changes, (b) merging, (c) change impact analysis, (d) change dependencies, and (e) understanding development tasks by means of questions. We discussed how their goals relate or differ from ours.

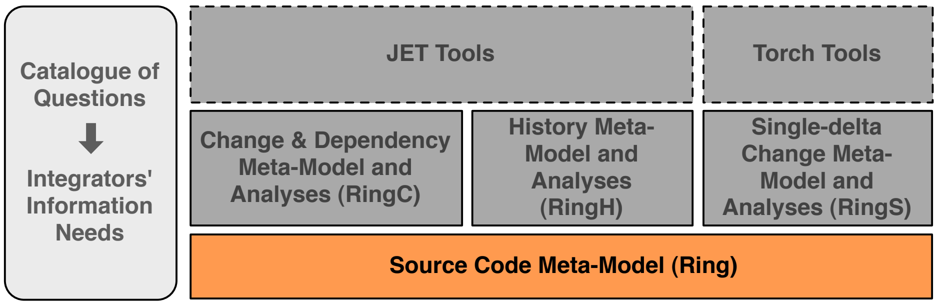
The next chapter introduces the core – the *Ring* source code meta-model – of the infrastructure needed to assess our contributions. Our history and change models and the analyses that will be presented in later chapters are built on top of it.

CHAPTER 4

Ring: a Unified Model for Source Code Representation

Contents

Contributions Map



Catalogue of Questions
Integrators' Information Needs
JET Tools
Torch Tools
Change & Dependency Meta-Model and Analyses (RingC)
History Meta-Model and Analyses (RingH)
Single-delta Change Meta-Model and Analyses (RingS)
Source Code Meta-Model (Ring)

Overview

This chapter presents the foundation of our infrastructure, an object-oriented model for source code representation, and the two analyses that serve as a motivation for such a meta-model. This source code model is a technical contribution of this dissertation. We have published this work in the *Journal of Computer Languages, Systems and Structures* [Uquillas Gómez 2012] and presented it at the *Smalltalks Conference* [Uquillas Gómez 2010a]. First, we present the requirements we established for source code modeling. Second, we present an analysis of data models manipulated by common version control systems as a means to identify how such systems relate to the actual source code of a system. Third, we present an analysis of several source code meta-models as a means to motivate the need of a unified and simple source code meta-model. Fourth, we describe the architecture of our object-oriented source code meta-model, namely *Ring*. Fifth, we present two concrete illustrations of the usage of our model applied to existing tools of the Pharo environment. Finally, we present a discussion of technical improvements to *Ring*.

4.1 Introduction

Version control systems such as Subversion or Git record different versions of code. They may differ in the way such data is stored, for example as deltas representing only changes or as snapshots of the system representing the complete version. This information is then accessible to other tools for specific tasks. For example, the source code can be accessed and processed for detecting changes between versions and providing conflict analysis as well as support elementary merging. Nowadays, there is little support out of the box to be able to perform queries and analyses over the complete history: Tools have to build their own infrastructure and history analysis on top of version control systems [Zimmermann 2004b]. For example, to compare all the differences between past senders of a given method is not straightforward. Another example is how to support cross-branch merging. Such examples, however, should be based on source code models [Lethbridge 2004]. Therefore, to ease history and change analyses we need adequate models to represent source code program entities. When considering such a representation, various design dimensions have to be taken into account, such as the level of granularity, the needed API, etc.

Based on two analyses performed on four version control systems data models presented in Section 4.3 and on seven source code models presented in Section 4.4, we established the requirements for the source code representation needed for our infrastructure. This code representation allows us to establish a foundation to provide integrators' information needs that we identified in Section 3.4 and that can support answering the integrators' questions (presented in the catalogue in Section 3.3.2). Our solution, the *Ring* source code meta-model, is described in Section 4.5 as a technical contribution of this dissertation. It is the basis for our change and history meta-models that we present in later chapters. Moreover, *Ring* is already part of the infrastructure of Pharo released with the Pharo version 1.4 in April 2012.

In the next section, we discuss the requirements for *Ring* and we motivate the reasons of why we did not reuse an existing meta-model.

4.2 Requirements for Source Code Modeling

The source code and the history of a system are a valuable resource for software engineers, developers and integrators [Lethbridge 2004]. They often need to analyze and understand the current source code or the evolution and changes of a system before performing actual maintenance or integration tasks.

The analyses of source code meta-models presented in Section 4.4, and especially referring to the Smalltalk-oriented meta-models, served us to identify that several source code meta-models coexist in a weakly causally connected way[1] [Maes 1987]: the Smalltalk structural and reflective API coexists

[1]Causal connection is defined by Maes as: "A computational system is said to be causally connected to its domain if the internal structures and the domain they represent are linked in such a way that if one of the two changes, this leads to a correspond effect upon the other"

with the one of the Refactoring Browser or with any of the two versions of the Monticello distributed version control system (MC1 or MC2). While having application-specific meta-models is an adequate engineering solution when developers want to abstract over different systems and be independent of idiosyncrasies of the underlying execution platform, in reality it multiplies the number of abstractions, it increases maintenance efforts and reduces tool reuse when in the presence of non-polymorphic APIs.

Furthermore, these meta-models offer a different API than the runtime and structural model of Smalltalk. We call this problem *the meta-models plague* that is, when multiple meta-models have different APIs which make the conversion between them, change propagation and test assessment difficult. This proliferation of meta-models puts the burden on the developer that has to maintain consistent models across tools. As a result developers resort to adding or modifying the same behavior (*e.g.,* introducing a predicate) in the different models for complying with the non-polymorphic APIs. We believe that this is due to the lack of a source code meta-model which could be extended and be the glue between source code models and tools as well as an adequate infrastructure [van den Hamer 1996].

We have also found that none of these models provide us with a simple and complete representation of the source program entities that we need to provide the integrators' information needs described in Section 3.4. They do not suffice for our specific needs, therefore, our primary requirement is to define our own source code meta-model.

Next to this primary requirement, we have identified several secondary requirements for building our source code meta-model that emerge from reuse and practical integration with the host environment, *i.e.,* the Pharo environment. These requirements are not related to the problem of assisted integration, and they do not represent a scientific contribution of this dissertation. However, they provide us with technical benefits for the foundation of our infrastructure.

No duplication of meta-models. We do not want one source code runtime meta-model and another for the change and versioning system. Having different meta-models is costly to maintain, test, and keep in sync. Our goal is to define a common source code core meta-model that can be extended for specific tasks. This may come at the cost of having some parts of the objects not used for certain scenario(s). To solve this problem, the entities within the meta-model should be able to be annotated with any additional information that is not defined beforehand.

Model update as cheaply as possible. Updating models is also a problem since desynchronization of the represented information may lead to subtle bugs. In addition, Smalltalk has its own reflective meta-model that is used by the runtime system [Black 2009] which is causally connected (meaning that the model reflects its subject in any circumstances). Therefore, the new model should use the causal connection as much as possible.

Tool reusability relying on common APIs. Currently it is common for new tools to define their own meta-model that provides a non-polymorphic API with respect to other models for representing entities. This hampers the reuse of tools manipulating source code entities, as they may have different APIs. Having a common meta-model will ease the integration and reusability of those tools.

Model integrated into the Smalltalk environment. Since we use Smalltalk as a testbed for the validation of this dissertation for the reasons presented in Section 1.3, the source code meta-model should be applicable to Smalltalk.

4.3 Version Control Systems Data Models

In Section 2.1.1 we introduced Version Control Systems (VCS) as systems that allow users to track and store changes, collaborate and share project files. We stated that version control systems support collaborative development by integrating approaches such as branching and merging. In particular, distributed version control systems (*e.g.,* Git) have given branching a central role to guide the development process.

Version control systems use a data model to store the source code of a system. The data model has an impact on how the information is processed. Most version control systems store and manipulate text files with changed source code *i.e.,* commits, or files representing the source code of the complete version. In both cases, they do not know the domain-entities they manipulate and are oblivious to the semantics of the system.

The relation between the source code model and the version control system data model might not be obvious. While the first models each entity as a first-class object, the latter mainly keeps track of files of source code and treats them as a plain text. The mismatch between the source model used by the IDE's tools (*e.g.,* package class browsers in Eclipse) and the data model used by VCSs leads to extra efforts to connect two different abstractions. Having this gap is already a reason for having different APIs and transformations between those two models.

While some specialized version control systems such as Monticello [Black 2009] or Envy [Thomas 1988, Pelrine 2001] (in the past) keep track of classes and methods, not all of them do so. Git [Chacon 2008] or Subversion [Collins-Sussman 2009] keep track of code; they also support binary files. Since there may not be a direct connection between the stored model and the actual source code, this results in a need to understand the data models used by version control systems and their links to the actual source code. For example using Git to directly manipulate methods/classes implies building an extra infrastructure as it stores objects with a chunk of binary data representing the files that contain the definitions of method/classes, but not those entities independently.

In the following, we present the most representative data models used by version control systems with a bias towards Smalltalk solutions. They are categorized in two groups: text-based and code-based data models.

4.3.1 Text-based Version Control Systems Data Models

We present the data models used by the Subversion centralized version control system and by the Git distributed version control system.

4.3.1.1 Subversion (SVN) Data Model

Subversion [Collins-Sussman 2009] is a centralized system for sharing information. At its core is a repository that stores data centrally in the form of a filesystem tree –a typical hierarchy of files and directories. Any number of clients connect to the repository, and then read from or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.

The Subversion repository remembers every change ever written to it – every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.

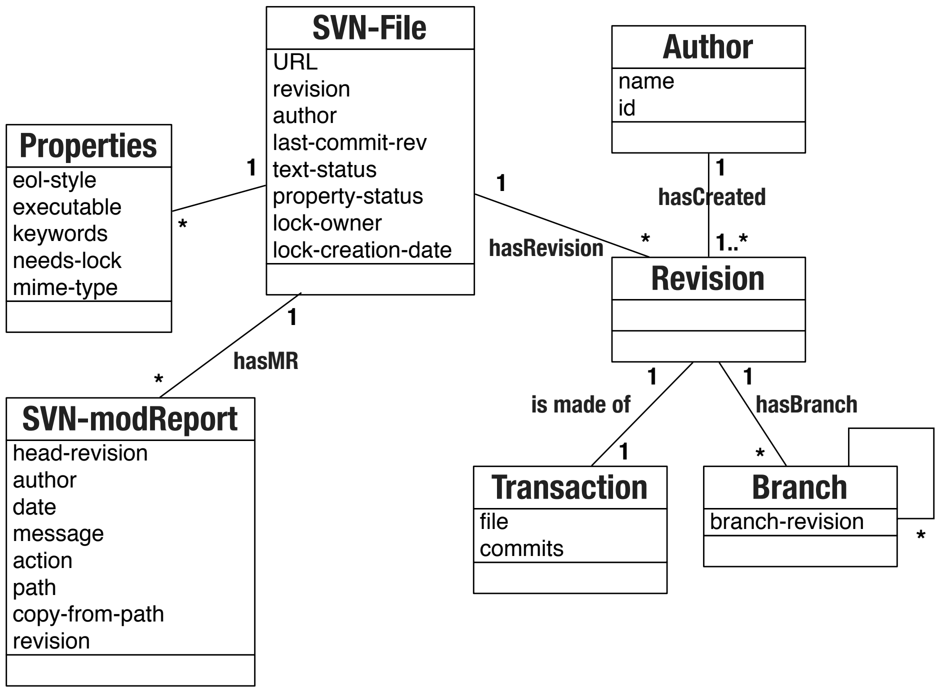


Figure 4.1: *The Subversion data model.*

The Subversion data model[2] is shown in Figure 4.1.

SVN-File. Subversion manages directories with files. However, it does not actually make a distinction between files or directories. Hence the data model only presents one definition, SVN-File, for both directories and files. The SVN-File keeps the following information:

- *URL*: the path to the file in a SVN repository
- *revision*: the most current revision of a file or directory
- *author*: the author to whom the file "belongs"
- *last commit revision*: the revision and the timestamp when the revision was committed
- *text status*: indicates whether the file has been modified locally or both locally and in the repository, added or removed
- *property status*: provides information about the non versioned properties of a file, transaction or directory tree (*i.e.,* the timestamp when the transaction was created)
- *lock owner*: the name of the person that made a lock (read, write) to the file
- *lock creation time*: the date and time when the lock was applied to the file

[2]The Subversion data model has been extracted from [Marjanovic 2006].

Branch. Subversion has no internal concept of a branch – it only knows how to make copies. When a directory is copied, the resultant directory is only a "branch" because we attach that meaning to it. Therefore, a SVN-File can have multiple branches, and they exist as normal filesystem directories in the repository. This is different from other version control systems, where branches are typically defined by adding extra-dimensional "labels" to collections of files. Finally, branches in SVN can have branches of their own.

Transaction. Unlike CVS, Subversion has a defined transaction concept. Transactions help in distinguishing a set of operations to a file that belong to a single development step as, for instance, a set of changes that lead to a new revision of a file. A transaction in SVN represents a set of commits that apply to a file before the current revision changes to a new one.

Author. The author information is represented as a separate entity because it is considered to be valuable information. An author entity holds the name of the author and an optional id, if present.

SVN-modReport. Unlike CVS, where the modification reports for a file are appended to the file log, SVN maintains the file and the modification information separately. When looking at the modification report log, it provides the particular action (modified, added, deleted), the timestamp, the author, etc. for each revision.

Properties. Properties in SVN designate the additional information in the form of tags or keywords to a file, and they are kept as separate entities. A SVN-File can have multiple properties set. The most interesting property is keywords. They are a common concept in many versioning systems, such as CVS.

4.3.1.2 Git Data Model

Git [Git 2005] is a distributed revision control system (DVCS). The core of Git is composed of a collection of tools that implement a tree structure storage and directory content management system [Chacon 2008].

Git differs from most version control systems such as Subversion, CVS, etc. in the way it stores data. These VCS systems store information as a list of file-based changes representing the code deltas or diffs between one commit and the next. Instead, when Git stores a new version of a project, it stores a snapshot of all the files in that project at a point in time. The snapshot is stored as a new tree – a bunch of blobs of content and a collection of pointers which a full directory of files and subdirectories can be recreated with. If a file has not changed, Git does not store the file again – only a link to the previous identical file it has already stored. In Git, the differences between two versions is calculated by running a new `diff` on the two trees representing both versions.

Git defines *objects* which represent the actual data. There are four main immutable object types that are stored in the Git *Object Database*, which in turn is kept in the Git *Directory*. Each object is referenced by a unique hash key (`SHA-1`) of its content plus a small header.

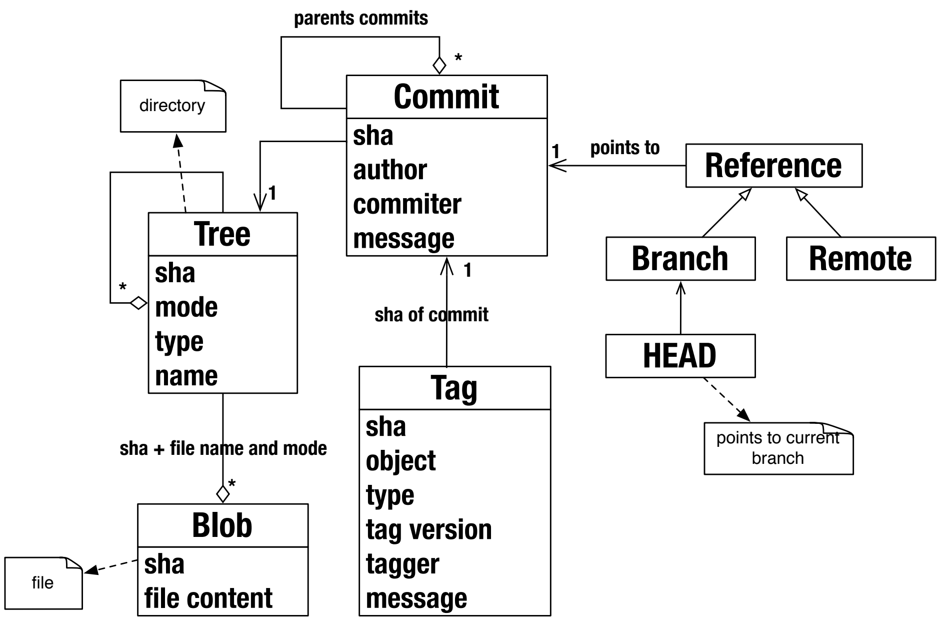


Figure 4.2: *The Git data model.*

Blob. The content of each file is stored as a *blob*. The files themselves –names and modes– are not stored within the blobs, just their content. Differently named files with the same content will only store one blob and share it. Therefore, during repository transfers (*i.e.,* clones or fetches) only one blob will be transferred, then expanded into multiple files upon checkout. The blob is totally independent from its location in the directory tree, and renaming a file does not change the blob that this file is associated with.

Tree. The physical directories map to *trees*. A tree is a simple list of pointers to blobs and other trees, along with the names and modes of those trees and blobs. The content section of a tree object consists of a very simple text file that lists the mode, type, name and hash key of each entry.

Commit. The tree history is managed by *commit* objects. A commit is similar to a tree. It points to a tree (representing the contents of a directory at a certain point in time) and keeps an author, committer, message and any parent commits.

Tag. Commits can be referred to by *tags*, *i.e.,* permanent shorthand names. A tag contains an object, type, tag version, tagger and a message. Normally the type is commit and the object is the hash key of the commit that is being tagged.

In addition to immutable objects, mutable *references* are stored in Git as well. A reference is a pointer to a particular commit, similar to a tag, but easily moveable. References are used for controlling branches and remotes.

Branch. A branch is just a file that contains the hash key of the most recent commit for that branch.

Remote. A remote is basically a pointer to a branch in another person's copy of the same repository (*e.g.,* by cloning a repository).

Figure 4.2 shows the data model with the objects and references stored in Git. Note an extra element, the HEAD file that points to the branch a developer using Git is currently working on, and that is used as the parent of the next commit.

4.3.2 Code-based Version Control Systems Data Models

Code-based version control systems keep complex entities to represent versioning information. We present the data model used by both versions of the Monticello distributed version control system.

4.3.2.1 Monticello 1 Data Model

Monticello 1[3] is a distributed concurrent versioning system for Smalltalk dialects such as Pharo, Squeak, GemStone and Cincom Smalltalk, in which classes and methods, rather than lines of text, are the units of change [Black 2009]. Monticello 1 (a.k.a. *MC1*) is organized around *snapshots* of a package, that are stored as *versions*. Snapshots are a declarative model of the Smalltalk code containing a package composed of classes and methods.

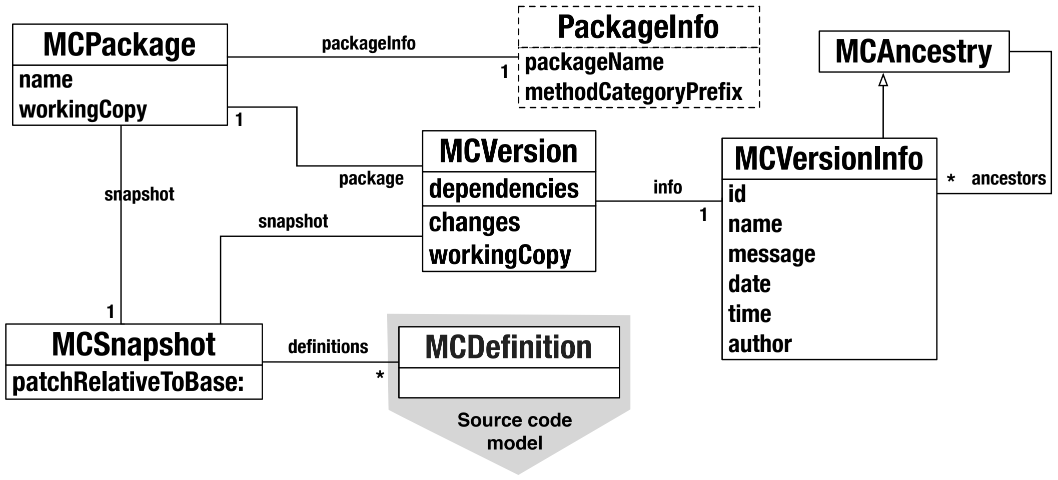


Figure 4.3: *Monticello 1 data model.*

We present an overview of the MC1 data model in Figure 4.3. The main entities of the data model are packages, snapshots, and versions.

Packages. A *package* is the unit of versioning. The classes and methods contained in a package are recorded and versioned together in a snapshot.

Snapshots. A *snapshot* is the state of a package at a particular point in time. It includes definitions of classes, methods, variables, traits[4] and package categories.

[3]Monticello 1: http://www.wiresong.ca/monticello/v1

[4]A trait is a set of methods that serves as a behavioral building block for classes [Schärli 2003, Ducasse 2006b]. Classes that use traits are still organized in a single inheritance hierarchy, but the traits specify an incremental difference in behavior with respect to their superclasses.

Versions. A *version* is a snapshot of a package. It also stores associated metadata such as VersionInfo and the version's ancestries. Versions are stored as zipped Monticello files `.mcz`, and represent the standard data used by the system.

In summary, MC1 records a series of snapshots of the code corresponding to a package as it evolves, as well as the ancestral relationships between snapshots. When loading a snapshot into an Smalltalk image, MC1 locates the differences between this snapshot and the state of its package in the image, and then makes the necessary changes to the image so that it matches the snapshot. It uses the ancestries of snapshots to provide a merge operation, so that conflicts between two sets of changes can be detected, and non-conflicting changes can be applied automatically.

The data model is connected to the source code model through snapshots as shown in Figure 4.3 with a grey background. In Section 4.4.2.4 we provide detailed information about the source code model.

In spite of the presented benefits, the data model that Monticello 1 uses is based on the assumption that packages are well-defined and have relatively stable boundaries (*e.g.,* packages are not expected to be removed or renamed, or their classes will not be moved to other packages), which is not always the case. In addition, Monticello 1 limits the history to the level of packages and not to the level of independent program entities. These issues are addressed by Monticello 2.

4.3.2.2 Monticello 2 Data Model

Monticello 2[5] (a.k.a. *MC2*) addresses the main problem encountered with Monticello 1, which is its unit of versioning – the package – that is too coarse-grained for many situations that arise in normal development (*i.e.,* changes may only impact a few methods or classes, but still the whole package needs to be versioned).

In Monticello 2, a new data model has been incorporated that does not have packages as the fundamental unit of versioning. Instead, the unit of versioning is individual program elements (*e.g.,* classes, methods, instance variables, and so on). This means that Monticello 2 can be used to version arbitrary snippets of code. These might correspond to packages, change sets, or any other method a developers chooses to separate “interesting" code from the rest of the image.

Rather than maintaining the version history of packages, Monticello 2 keeps track of the version history for each element. Having such a history allows users to perform tasks that are not possible with Monticello 1. With this data model package boundaries are no longer a restriction. Packages can be created, renamed or destroyed, elements can be moved back and forth between packages, elements can even belong to more than one package at a time. Since the version history is attached to the element, it is not affected.

In Figure 4.4 we show the main classes of the MC2 data model. Note that ancestry information is now linked at an entity-based level (ElementVersion) and not at the level of the package as in Monticello 1.

Elements. An *element* is a representation of a specific program entity (*e.g.,* classes, methods, variables).

[5]Monticello 2: http://www.wiresong.ca/monticello/v2

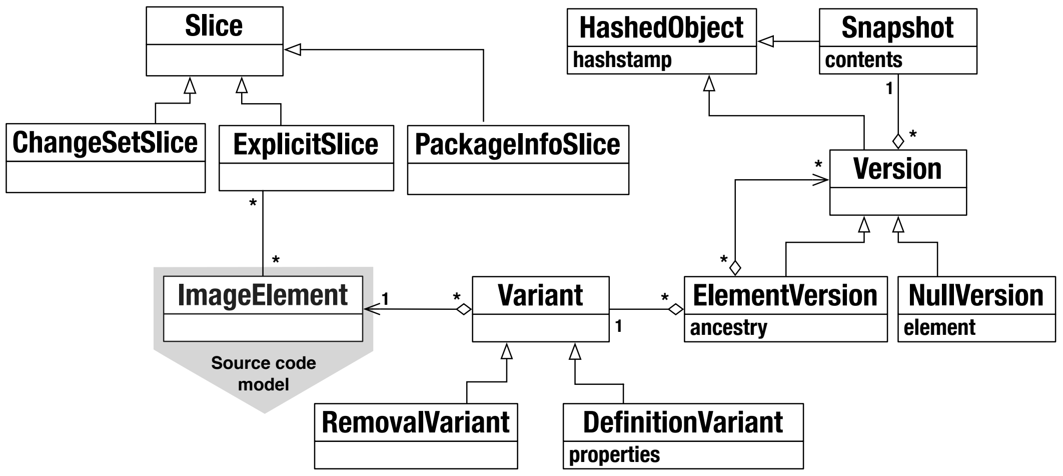


Figure 4.4: *Monticello 2 data model.*

Variants. A *variant* describes the state of a particular element.

Versions. A *version* represents the state of an element at a particular point in time. A version associates a variant of an element with the ancestry of that element (*i.e.,* set of versions that precede this version), and it is identified by a hashstamp.

Hashstamps. A *hashstamp* is a unique identifier given to each version.

Slices. A *slice* groups elements together. Slices are independent and can overlap. Elements can belong to many slices at the same time or to none. Different types of slices are supported: PackageInfoSlice for elements defined in a given package, ChangeSetSlice for elements associated with a given ChangeSet, and ExplicitSlice for a particular collection of elements.

Snapshots. A *snapshot* captures the state of a slice. Snapshots record the versions' hashstamps of the slice's elements.

The MC2 data model (element-based version history) allows developers to merge changes of individual elements. Although MC1 supports cherry-picking, it does so in an awkward and non-intuitive way. In MC2, cherry-picking is the norm, and merging an entire package is just a special case. However, MC2 does not help developers finding any semantical issue when cherry picking and merging.

The data model is connected to the source code model through variants and explicit slices as shown in Figure 4.4 with grey background. The source code model is presented in Section 4.4.2.5.

4.4 Dedicated Source Code Meta-Models

While versioning focuses on how to merge and version between versions, it is important to look at source code models. If we take for example Smalltalk, there are several source code meta-models with different purposes (*e.g.,* for managing changes, refactorings, merges and versions) that manipulate in some way the Smalltalk structural meta-model. Most of the time, such meta-models are overlapping or included in each other. This overlap often exists for a good reason. For example, the Refactoring Engine was developed in VisualWorks and should work on any other Smalltalk dialect, therefore the authors preferred to extract and build their own representation instead of extending the existing one. A similar concern exists for Monticello.

Another important concern that we should pay attention to is that Smalltalk is a reflective language [Rivard 1996, Ducasse 1999]. This means that it has a causally connected representation of itself [Maes 1987]. Such a causal connection between the model of Smalltalk and its execution is a powerful mechanism that supports tool building. When new models are populated to represent views of the Smalltalk runtime, the question of the causal connection is key: "Should tool builders recreate the model each time the runtime changes?", "How do they maintain consistency across models?". For example, in the Moose[6] software analysis platform, a model is created for a version or for the actual code, but if such code changes the model needs to be recreated. Moose keeps immutable models as it focuses on being able to manipulate source code written in different languages; Smalltalk being one among others (Java, C, C++) [Nierstrasz 2005]. But since Moose is implemented in Smalltalk, it could be possible that for a single version analysis we could use the casual connection to the actual source code, and avoid recreating the model when changes happen.

In the following, we present the most representative source code meta-models which can serve as an inspiration for a new source code meta-model. We analyzed these source code meta-models to identify how they represent domain entities within an object-oriented software system such as classes, packages, etc. Our goal was to take the best features of each model to incorporate into our *Ring* source code meta-model described in Section 4.5. We classify these meta-models into two groups: non-Smalltalk specific and Smalltalk-oriented code meta-models.

4.4.1 Non-Smalltalk Specific Code Meta-Models

We present the Eclipse Modeling Framework along with the ECore meta-model to represent Java systems, and the language-independent FAMIX core meta-model to represent system written in Java, C, C++, and Smalltalk.

4.4.1.1 Eclipse Modeling Framework (EMF)

According to the EMF website[7], "it is defined as a modeling framework and code generation facility for building tools and other applications based on a structured data model". Meta-models can be specified using annotated Java, XML documents, UML, or modeling tools like Rational Rose, then imported into EMF. The specification of a meta-model described in XMI (XML Metadata Interchange) serves to provide tools and runtime support that can generate a set of Java classes, such classes represent an EMF model.

[6]Moose: http://www.moosetechnology.org
[7]EMF: http://www.eclipse.org/modeling/emf/?project=emf

ECore and GenModel. ECore[8] is a meta-model for describing models and run-time support for the models. EMF provides two ways for instantiating models that conform to an ECore meta-model: by reflection or by code generation. For the latter, the Java classes are generated from an ECore meta-model specification. This process is performed in two steps: (1) the ECore meta-model is transformed into a GenModel[9] model that can contain additional implementation-specific information. (2) a model-to-text (M2T) transformation consumes the GenModel in order to generate the functional Java code.

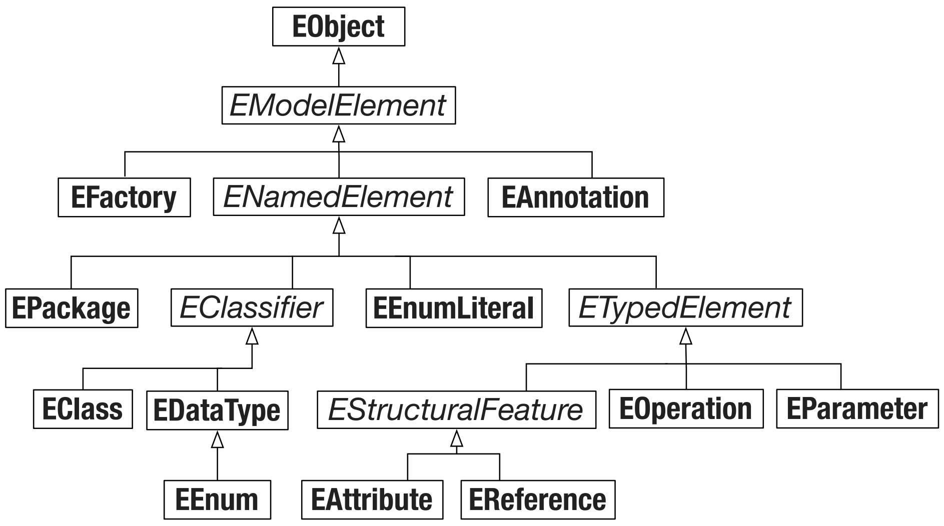


Figure 4.5: *EMF Ecore code meta-model – Definitions that appear in italic represent features otherwise elements of the meta-model.*

The Ecore meta-model is shown in Figure 4.5. It allows developers to define different elements such as classes, packages, parameters, attributes, etc. According to the ECore specification [Vogel 2012], several of these meta-model elements are defined as:

- **EClass**: represents a class, with zero or more attributes and zero or more references.
- **EAttribute**: represents an attribute which has a name and a type.
- **EReference**: represents one end of an association between two classes. It has a flag to indicate if it represents a containment, and a reference class to which it points.
- **EDataType**: represents the type of an attribute, *e.g.,* int, float or java.util.Date.
- **EOperation**: represents a method which has a name, a return type and may have parameters as input.
- **EParameter**: represents a parameter of a method. It has a name, type and multiplicity as input.

[8]ECore: http://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/ecore/package-summary.html
[9]GenModel: http://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/codegen/ecore/genmodel/package-summary.html

4.4.1.2 FAMIX

FAMIX 3.0[10] is a family of meta-models for software analysis and various aspects of code representation (static, dynamic, history). These models were developed in the context of the Moose software analysis platform [Nierstrasz 2005]. The meta-models are implemented in Smalltalk, and provide a rich API that can be used for querying and navigating. The core of FAMIX [Demeyer 2001] is a language independent meta-model that provides a generic representation of the static structure of programs written in multiple object-oriented and procedural programming languages, such as Smalltalk, Java, C, and C++.

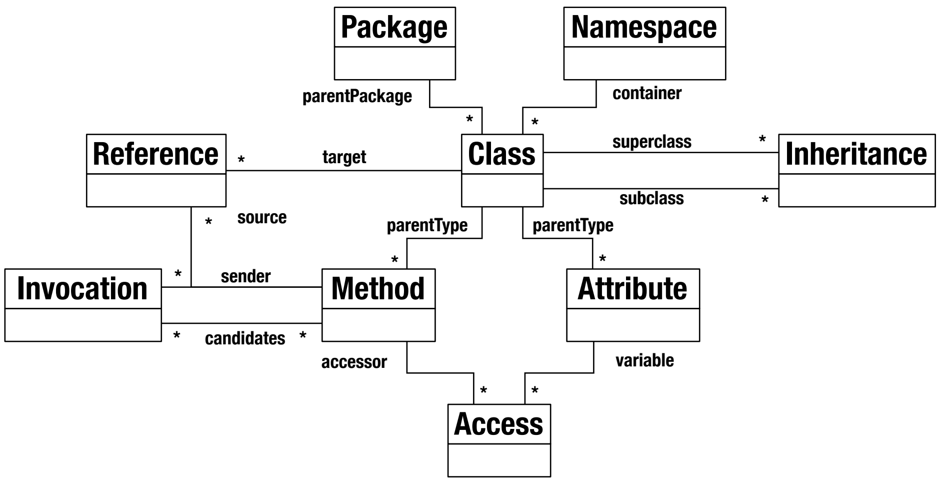


Figure 4.6: *FAMIX-core language independent code meta-model - Key Classes.*

The core meta-model consists of a set of classes that represent source code at the program entity level. Such classes map onto the different elements in a program (*e.g.,* classes, methods, attributes, comments), and of the associations between these elements (*i.e.,* inheritance definitions, invocations of methods, accesses to attributes by methods, references to classes by methods). Figure 4.6 shows the FAMIX-core code meta-model. While the meta-model is fairly complete, it can be easily extended in order to incorporate other language extensions.

Key points. There are two important points in the design of FAMIX that are worth stressing:

1. FAMIX does not only represent structural source code entities such as *packages, classes, methods* but it also explicitly represents information that is extracted from the methods' abstract syntax trees: a method *refers* to a class (Reference), a method *accesses* attributes (Access) and a method *invokes* other methods (Invocation). In this way, FAMIX offers a finer-grained representation of a program than a simpler meta-model and it does so in a language independent manner. Fact extractors, which by definition have the knowledge of the targeted language, produce language independent information in terms of FAMIX models.

2. FAMIX provides decoupling between *packages* and *namespaces*. Namespaces are scoping entities that provide a lexical scope for the contained entities, while packages are scoping entities that describe the physical structure of a system (*i.e.,* deployment entities). This decoupling makes sure that FAMIX can model any kind of situation at the package level.

[10]The Moose book: http://www.themoosebook.org/book/internals/famix

Our approach does not use the FAMIX core meta-model for three reasons:

1. FAMIX is too tainted with language independent features, *e.g.,* multiple inheritance types, interfaces, and lacks Smalltalk trait definitions.

2. We had specific needs such as to only represent what is needed to provide the integrators' information needs in order to assist the integration process. Therefore, we opted for defining our own source code meta-model, inspired by the FAMIX core meta-model.

3. We targeted a unified source code meta-model, namely *Ring* [Uquillas Gómez 2012], that can be used as the foundational code model for tool integration in Pharo. Therefore, we built *Ring* as a technical contribution for this dissertation.

4.4.2 Smalltalk-oriented Code Meta-Models

We present five Smalltalk source code meta-models: the Refactoring Browser code scoping meta-model, the Smalltalk runtime and structural model-model, the Ginsu semantic meta-model, and the code meta-models of both Monticello 1 and Monticello 2 version control systems.

4.4.2.1 Refactoring Browser

The Refactoring Browser[11] (*RB*) [Brant 1998, Roberts 1997, Roberts 1999] is a powerful Smalltalk browser which enables developers to perform several automated refactorings on Smalltalk programs, such as pushing up methods, renaming variables, splitting classes, etc. The refactorings can be classified into three groups: class based refactorings, method based refactorings, and code based refactorings. RB also offers other productivity enhancements for programmers: *Smalltalk Code Critics*, a tool that analyzes code for detecting bugs or possible errors; and the *Rewrite* tool for expressing the rewriting of code through recognition of expressions (pattern matching) on ASTs.

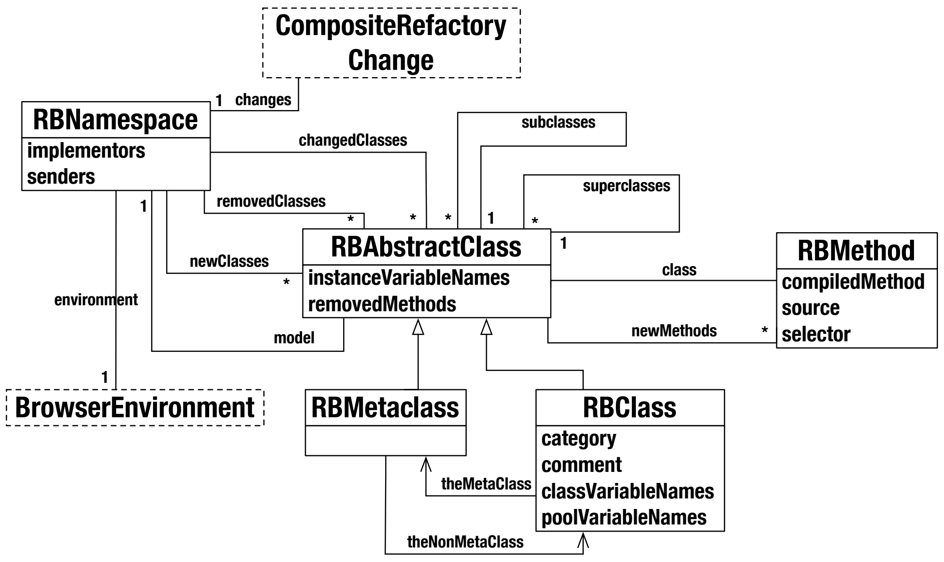


Figure 4.7: *Refactoring Browser source code scoping model.*

[11]The Refactoring Browser: http://www.refactory.com/RefactoringBrowser

RB defines different models, each having a particular purpose. The following three models are the main ones: (a) the *refactoring model* represents specific refactoring operations; (b) the *changes model* represents changes associated with refactorings; (c) the *source code scoping model* – which is relevant to our approach – identifies the program elements that are manipulated for the rest of RB models. In addition, the *source code scoping model* models a delta with respect to the current system and is supposed to be polymorphic with the Smalltalk runtime and structural meta-model.

The complete source code model is shown in Figure 4.7. Two classes defined in another component of RG are BrowserEnvironment and CompositeRefactoryChange (shown with dashed border), both classes are associated with RBNamespace. The first represents the environment in which a namespace is defined and the second allows a namespace to control changes and refactorings.

This meta-model is a very simple model, only mapping classes, methods and namespaces. However, other elements such as variables or class comments are not modeled as first-class objects. Therefore, it does not suffice our requirements for source code representation.

4.4.2.2 Smalltalk Runtime and Structural Meta-model

Smalltalk itself defines a meta-model for representing entities at structural and runtime level [Goldberg 1989]. An excerpt of this meta-model extracted from Pharo is shown in Figure 4.8.

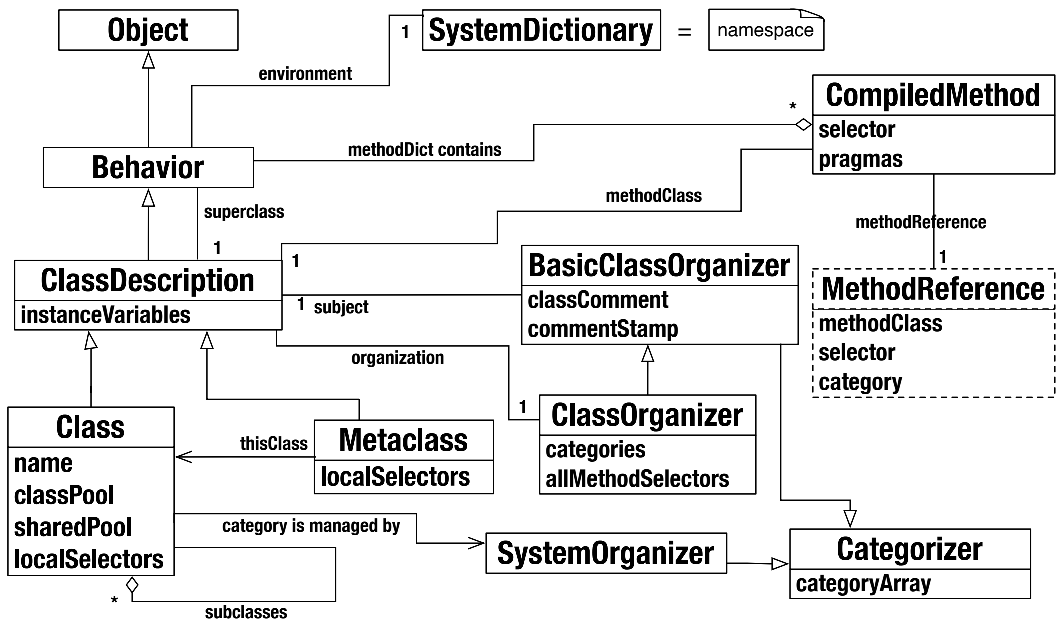


Figure 4.8: *Smalltalk (Pharo) structural code model (with dashed border an attempt to add a representational object for CompiledMethod).*

The main root class in Smalltalk is Object which defines common behavior for the rest of the classes. Classes and metaclasses derive from ClassDescription where instance variables are maintained in an array. Classes' methods are kept in a suitable form for interpretation by the virtual machine (*i.e.,* instances of CompiledMethod) and contained in a dictionary (methodDict). Classes are organized in categories, or what is commonly known as packages. However, this model only keeps category names in SystemOrganization, an instance of SystemOrganizer. The protocols of a class (*i.e.,* method categories) are managed by ClassOrganizer. Finally, every entity is associated with the environment (*i.e.,* namespace) in which it is known. This environment is unique and is represented by an instance of SystemDictionary.

Key points. There are three points to underline about the Smalltalk structural code model:

1. The model is causally connected with its execution. Therefore, there is no problem related to the synchronization of the model when a runtime entity changes.
2. The model is influenced by the information mandatory for the language execution. For example, instance variables are not first-class objects but just strings. This is a problem when we need to map meta-models targeted at program representations or versioning.
3. Figure 4.8 shows the class MethodReference that can be considered as a workaround to support a representation of compiled methods. This was needed to support tools browsing different versions of a method.

4.4.2.3 Ginsu

Ginsu[12] is a *cross-dialect* semantic model and toolkit for partitioning Smalltalk code into packages. Each package should have a clearly defined scope and prerequisite structure. One of the goals of Ginsu is to be able to build analyses about code that is not executing or living in a Smalltalk runtime image [Black 2009]. This goal is similar to the one of FAMIX but without the language independent aspect.

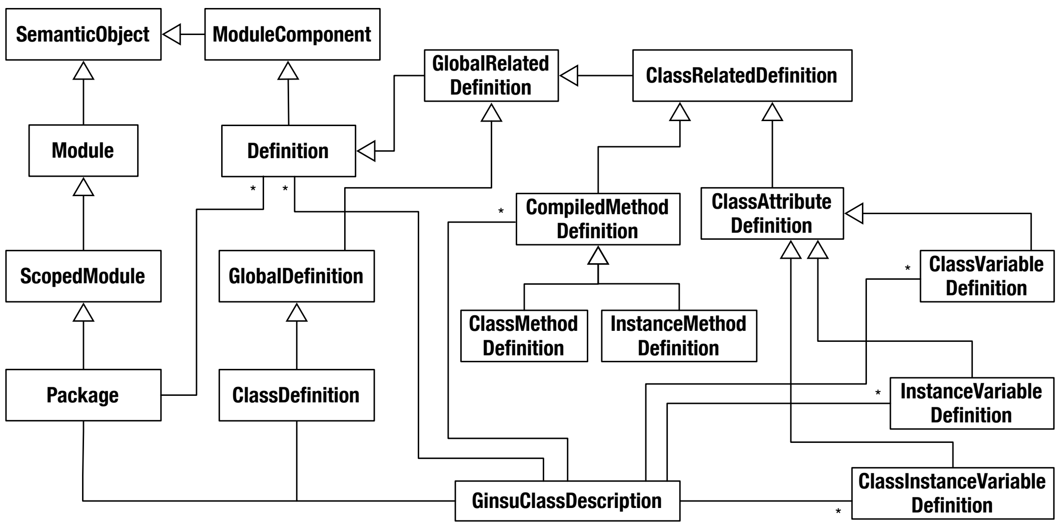


Figure 4.9: *Ginsu semantic model - Key classes.*

Ginsu maps the elements defined in Smalltalk code to semantic objects as can be seen in Figure 4.9. A semantic object represents the semantics of a Smalltalk program. Semantic objects (SemanticObject) are categorized as modules or components (subclasses of Module and ModuleComponent). Packages are mapped to modules, and the rest of the elements (*e.g.,* classes, methods, variables, etc.) to components. A particular definition (such as: ClassDefinition, InstanceMethodDefinition, ClassVariableDefinition, etc.) exists for each kind of component.

[12]Ginsu: http://sourceforge.net/projects/ginsu

The key classes defined in the semantic model are shown in Figure 4.9. Note that a package contains a set of definitions, the key idea behind Ginsu. An interesting property of Ginsu is its ability to annotate any semantic object. Annotations are easily maintained in a dictionary attached to each semantic object. In addition, the model defines the GinsuClassDescription which is associated with a class definition, a set of definitions, and a package. The Ginsu browsers (*i.e.,* PackageSystem and PackageSupport browser) interact with class descriptions.

Another interesting property of Ginsu is that when a semantic object is built for an entity that exists in the runtime (*i.e.,* image), Ginsu delegates all queries to the runtime object. This approach tries to get as much as possible out of the natural causal connection of the underlying Smalltalk model.

Ginsu lacks first-class representations for method associations, such as method calls, class references and access to variables, or the class inheritance relationship defined in FAMIX. In our approach, we leverage the idea of having objects with annotations.

4.4.2.4 Monticello 1 Source Code Model

The Monticello 1 distributed concurrent versioning system was introduced in Section 4.3.2.1. In this section, we present the source code model used by MC1. This model basically consists of definitions representing program entities.

Figure 4.10 shows the key entities of the source code model and how they are connected to the data model (shown with grey background) described in Section 4.3.2.1.

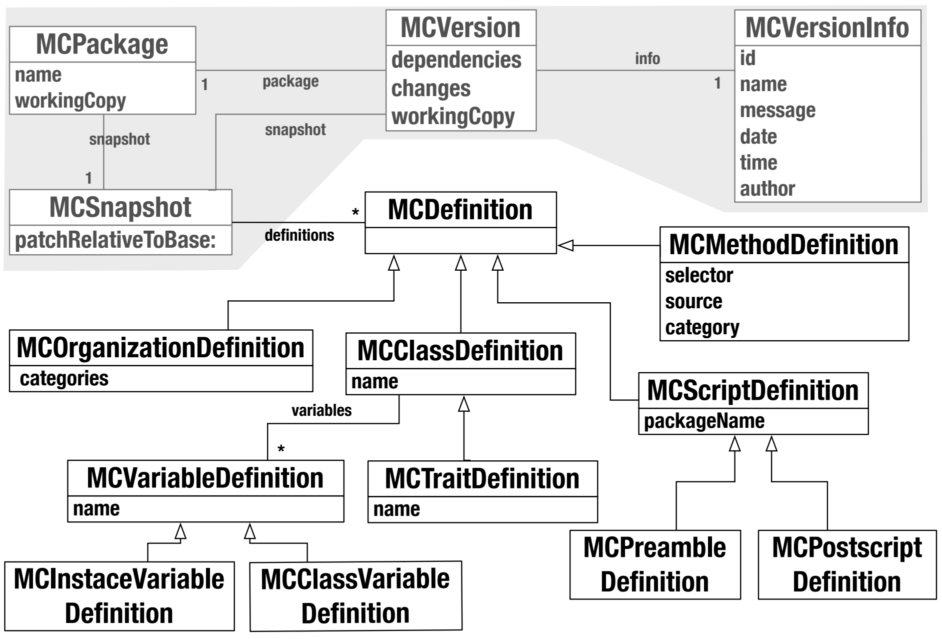


Figure 4.10: *Monticello 1 source code model – Key classes for program entities (in grey the data entities).*

A source code entity *definition* represents a program element (*i.e.,* class, method, variable, trait, package category, script). The source code model is composed of several classes. MCClassDefinition represents a class contained in a package. MCOrganizationDefinition represents the package's categories in which classes are contained. Subclasses of MCVariableDefinition represent variables of classes, and they are accessed by class references. MCMethodDefinition maps structural data of methods (selector, source code). Finally, MCScriptDefinition subclasses represent the pre/post conditions required by packages.

The MC1 source code model is not complete. For example, there are no definitions to represent class extensions[13] as first-class objects, but instead a naming convention in method categories (*i.e.,* protocols) is used. Moreover, this model is non-polymorphic with the API of the Smalltalk structural model presented in Section 4.4.2.2.

4.4.2.5 Monticello 2 Source Code Model

In this section we present the source code model of the Monticello 2 distributed concurrent versioning system introduced in Section 4.3.2.2.

Figure 4.11 shows the key classes defined in the source code model of MC2. It also shows the link of such classes to the data model presented in Section 4.3.2.2 (shown with grey background). Note that the data model accesses the source code elements through slices and variants.

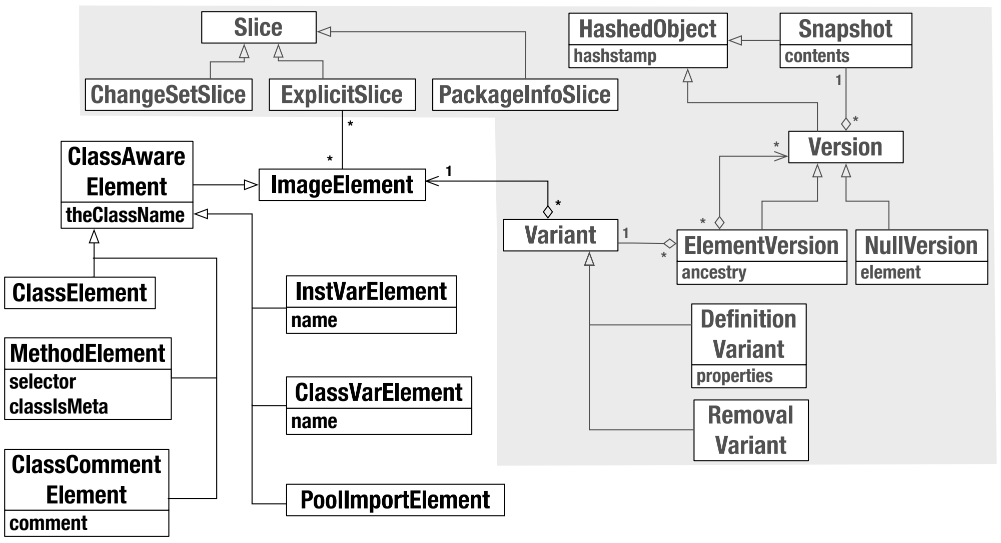


Figure 4.11: *Monticello 2 source code model – Key classes for program entities (in grey the data entities).*

The program elements are modeled by classes that inherit from ImageElement. In Monticello 2, an *element* is finer-grained than a *definition* in Monticello 1. For example, a comment is also represented as an element (ClassCommentElement). Elements are mostly related to a class and thus are defined as subclasses of ClassAwareElement. Class elements (*e.g.,* variables, methods) can be referred to directly, rather than by implication of the class reference.

This model also suffers from the same problem as the source code model of Monticello 1. It is non-polymorphic with the API of the Smalltalk structural model presented in Section 4.4.2.2.

[13]Smalltalk supports class extensions [Bergel 2005], *i.e.,* developers are able to add methods to classes in packages different from the ones which the classes belong to. This is a simple mechanism that allows developers to add behavior to existing entities without subclassing them in other packages.

4.5 The Ring Source Code Meta-Model

In Section 4.2 we discussed the requirements for source code modeling. Our primary requirement is to define a simple source code meta-model that allows us to represent the integrators' information needs or that serves as a base to obtain such information. We described these kinds of information in Section 3.4 as descriptive, structural, semantic and historical information.

From the source code representation of a system we can provide several descriptive and structural pieces of information such as authorship, structure or kind of entities. Note however, that the rest of information is obtained from the history and changes of a software system. Therefore, our source code representation should be the underlying model of the history and change models.

In this section, we stress the importance of getting a well-designed source code meta-model that serves as foundation for solid history and change meta-models. The key points of our source code meta-model are threefold.

- *Common API with the Runtime and Structural Smalltalk model*: allow existing and new tools to interact and integrate directly with the host environment, *i.e.,* Pharo.
- *Complete representation of program entities*: allow the definition of every program entity as first-class objects.
- *Extensive meta-model*: allow other models to use, refer or extend it, such as the history or change meta-models.

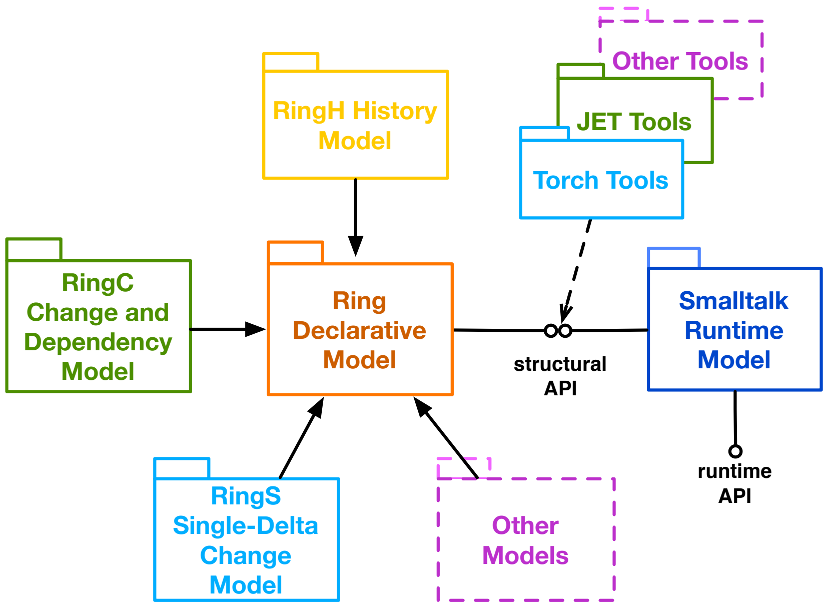


Figure 4.12: *The Ring overview.*

In Figure 4.12 we present an overview of *Ring*[14], our solution for the source code meta-model that is the base of our infrastructure. This figure shows how the components of our infrastructure and tools interact. Note that the declarative and runtime models share a common API which can be referred to by basic tools. This eases the reuse of such tools, for example a code browser should browse entities (*e.g.,* classes, methods) loaded in image or in external code files (*e.g.,* changesets).

[14]Ring: http://www.squeaksource.com/Ring

We show three concrete meta-models that extend *Ring*. The *RingS* single-delta change model is described in Chapter 5. The *RingH* history model and the *RingC* change and dependency model are described in Chapter 6. Regarding tools, we show the *Torch* tools that are explained together with *RingS* in Chapter 5, and the *JET* tools that are explained in Chapter 7. Note that we also show with a dashed purple border other models that can extend *Ring* such as a versioning model, and other tools that can use these models such as the version control system. The definition of such models and tools lies outside the scope of this dissertation and is considered future work.

4.5.1 Architecture of Ring

We designed *Ring* after analyzing the source code models presented in Section 4.4. From several of these models we took specific features that we consider beneficial for our approach. Concretely, we defined in *Ring* the concept of annotations provided by Ginsu (described in Section 4.4.2.3) as a means to add extra information to an object without affecting its structure. The FAMIX core meta-model described in Section 4.4.1.2 also provided us with features that were introduced in *Ring* such as the methods associations (*i.e.,* references to classes, accesses to variables, and invocations to methods). They were introduced to support fine-grained information at the history and change level. From the Refactoring Browser source code scoping model (described in Section 4.4.2.1) and from the Smalltalk structural model (described in Section 4.4.2.2) we took the notion of maintaining metaclasses as separate definitions.

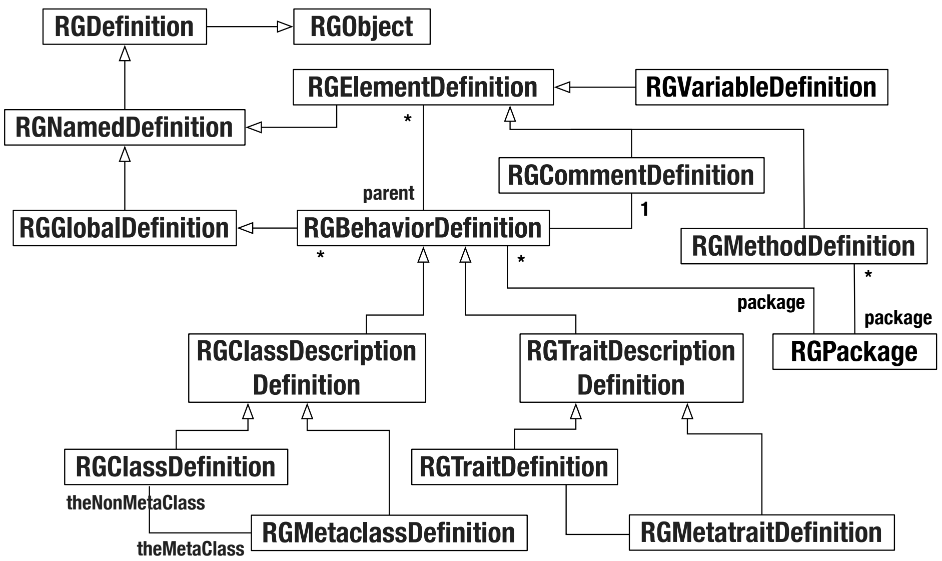


Figure 4.13: *Ring source code meta-model – Key definitions.*

In Figure 4.13 we present the main definitions of the *Ring* source code meta-model to provide fine-grained information that serves as the foundation to assist integration.

Base definitions. The root class in the source code meta-model is represented by RGObject. Every object in a *Ring* model is derived from this definition which provides support for annotations. Program entities such as classes, methods, variables, etc. are represented by definitions that subclass RGDefinition. It provides a link to a default environment (*i.e.,* namespace) in which each entity is known. Global program entities such as classes, metaclasses and global variables derive from RGGlobalDefinition. Finally, as many definitions are identified by a name we define the RGNamedDefinition that serve as indirect superclass.

Classes and Traits. The specific behavior of classes and traits for managing their definition, superclass, methods and protocols is supported by the RGBehaviorDefinition class. By means of this class we also provide the causal connection from a definition in the model to its corresponding existing object in the runtime environment (*i.e.,* image). The instance variables that can be defined in a class or metaclass are managed by RGClassDescriptionDefinition. This class is the direct superclass of both RGClassDefinition and RGMetaclassDefinition. For traits and metatraits we also defined a similar class hierarchy by means of the class RGTraitDescriptionDefinition which manages their client users (*e.g.,* classes). The concrete definitions of traits and metatraits is represented by RGTraitDefinition and RGMetatraitDefinition.

Elements of Classes and Traits. A class, trait or metaclass may consist of methods, variables, protocols and a comment. For representing these definitions we provide the abstract RGElementDefinition class. A class knows which are its elements, and each element knows which is its parent (*i.e.,* the class in which it is defined). An element within the model may also know its corresponding existing object in the runtime environment.

Class comments, methods and variables are defined by RGCommentDefinition, RGMethodDefinition and subclasses of RGVariableDefinition, respectively. A comment knows by whom and when it was written. A method knows several properties such as its protocol, source code, package, timestamp and author. Moreover, a method provides the causal connection to the runtime model by accessing the actual compiled Smalltalk method object, provided by the Smalltalk runtime environment.

Variable definitions. In *Ring* we support four kinds of variables that are shown in Figure 4.14 together with their relationships.

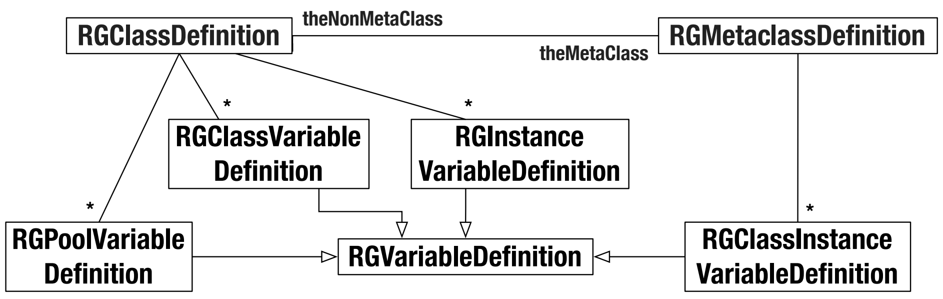

RGClassDefinition
theNonMetaClass
theMetaClass
RGMetaclassDefinition
*
RGClassVariable Definition
*
RGInstance VariableDefinition
*
RGPoolVariable Definition
RGVariableDefinition
*
RGClassInstance VariableDefinition

Figure 4.14: *Ring source code meta-model – Variables.*

Variables within Smalltalk can be defined in a class or metaclass. As seen in the figure, instance variables, class variables and pool variables may be associated with a class. They are defined by means of the classes RGInstanceVariableDefinition, RGClassVariableDefinition and RGPoolVariableDefinition respectively. A metaclass may define instance variables which are represented by RGClassInstanceVariableDefinition.

Container definitions. *Ring* also provides several definitions that model containers as shown in Figure 4.15. Even though only the RGPackage class is relevant for other definitions in the source code model, the other containers were defined to support the models and analyses built on top of *Ring*.

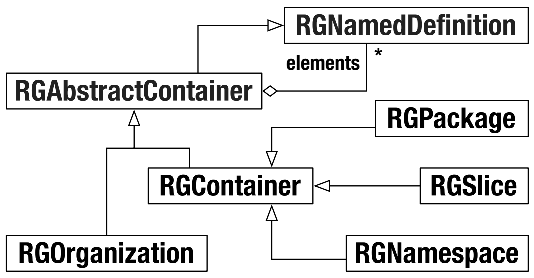


Figure 4.15: *Ring source code meta-model – Containers.*

RGAbstractContainer is the root definition of the container class hierarchy. It knows the elements that it contains and provides other definitions along with an API to manipulate such elements. RGContainer is the direct superclass of concrete containers (*i.e.,* namespaces, packages, slices), and provides the specific API to interact with three kinds of elements: classes, methods, and packages. Namespaces, packages and slices are represented by the definitions RGNamespace, RGPackage, and RGSlice respectively. A namespace represents the environment in which entities such as classes, traits, global variables and pool dictionaries are known. A package may contain classes, traits, class extensions and package categories. A slice is a set that groups entities from one or multiple packages, *i.e.,* it contains packages, classes and methods.

Within Monticello, packages are not a first-class entity but rather a property of a class (*i.e.,* package category). To accommodate this, we also include an organization container that enables analyses of Monticello data.

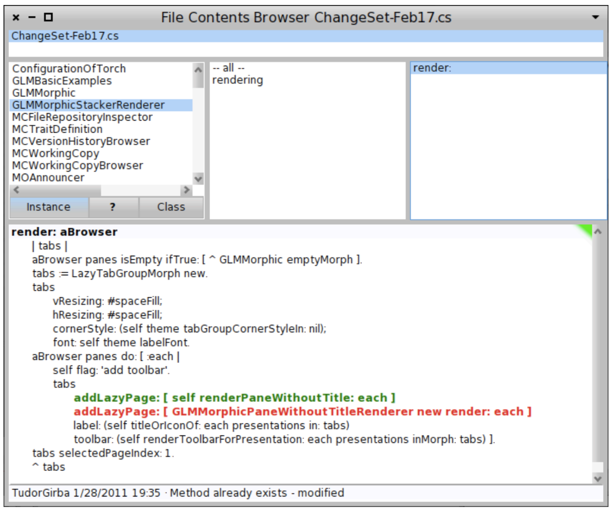
4.6 Ring Usage Scenarios

The *Ring* source code meta-model was picked up by the community and it is already integrated in Pharo 1.4 as the source code model for tool integration. In this section, we present two usage scenarios to demonstrate how simple it is to build tools on top of *Ring*. We show how two existing tools of the Pharo environment were ported to use *Ring* as their source code meta-model. Concretely, we present the *external code file browser* and the *refactoring browser source scoping model* using *Ring*. Note however, that both cases are under evaluation for future integration with Pharo.

4.6.1 External Code File Browser: Out-of-Image Code Browsing

Smalltalk IDEs allow developers to browse and load the contents of external source code files – change set files (`.cs`) or Smalltalk source files (`.st`). Before loading source code files into an image, developers usually browse the contents of such files to be sure that these files contain the needed source code.

For this task, the Pharo environment provides users with a FileContentsBrowser shown in Figure 4.16 (left). The source code is represented with a meta-model created for this particular browser.


File Contents Browser ChangeSet-Feb17.cs

ChangeSet-Feb17.cs

```
ConfigurationOfTorch
GLMBasicExamples
GLMMorphic
GLMMorphicStackerRenderer
MCFileRepositoryInspector
MCTraitDefinition
MCVersionHistoryBrowser
MCWorkingCopy
MCWorkingCopyBrowser
MOAnnouncer
```

```
-- all --
rendering
```

```
render:
```

```
render: aBrowser
    | tabs |
    aBrowser panes isEmpty ifTrue: [ ^ GLMMorphic emptyMorph ].
    tabs := LazyTabGroupMorph new.
    tabs
        vResizing: #spaceFill;
        hResizing: #spaceFill;
        cornerStyle: (self theme tabGroupCornerStyleIn: nil);
        font: self theme labelFont.
    aBrowser panes do: [ :each |
        self flag: 'add toolbar'.
        tabs
            addLazyPage: [ self renderPaneWithoutTitle: each ]
            addLazyPage: [ GLMMorphicPaneWithoutTitleRenderer new render: each ]
            label: (self titleOrIconOf: each presentations in: tabs)
            toolbar: (self renderToolbarForPresentation: each presentations inMorph: tabs) ].
    tabs selectedPageIndex: 1.
    ^ tabs
```

TudorGirba 1/28/2011 19:35 · Method already exists - modified

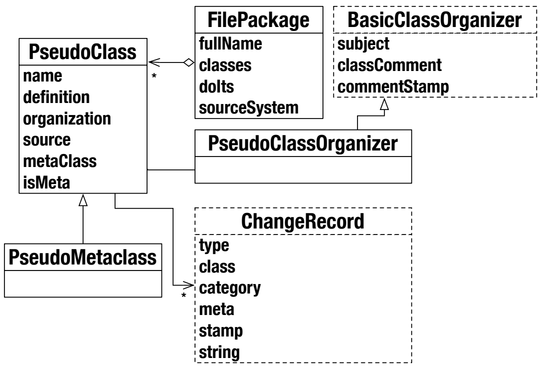


Figure 4.16: *The current file contents browser and its code meta-model.*

This model is known as the *pseudo classes model* and is shown in Figure 4.16 (right).

PseudoClass and PseudoMetaclass represent classes and their metaclasses. On the one hand, both classes are not related to the ones that define classes in the Smalltalk structural meta-model (shown in Section 4.4.2.2), and do not fully implement the same API. On the other hand, a part of the class data (*i.e.,* comment, stamp and method categories) is managed by PseudoClassOrganizer, a subclass of BasicClassOrganizer that is defined in the Smalltalk structural meta-model. Note that there is no pseudo-method definition, but instead ChangeRecord objects representing methods are associated to a pseudo class. This model also lacks a representation to explore traits from source code files.

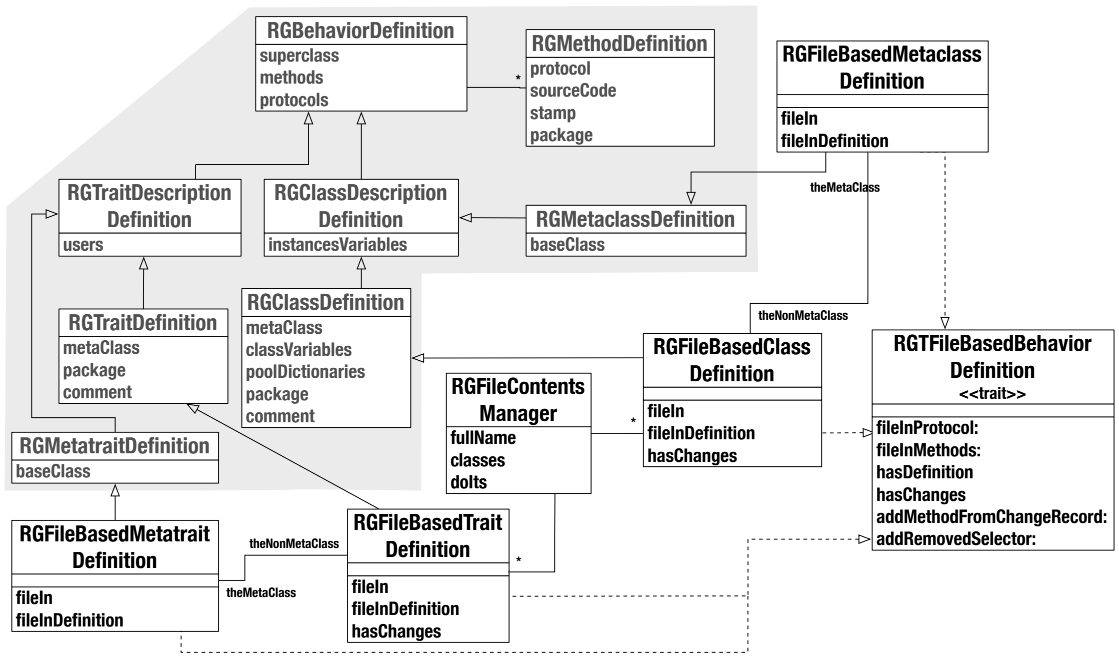
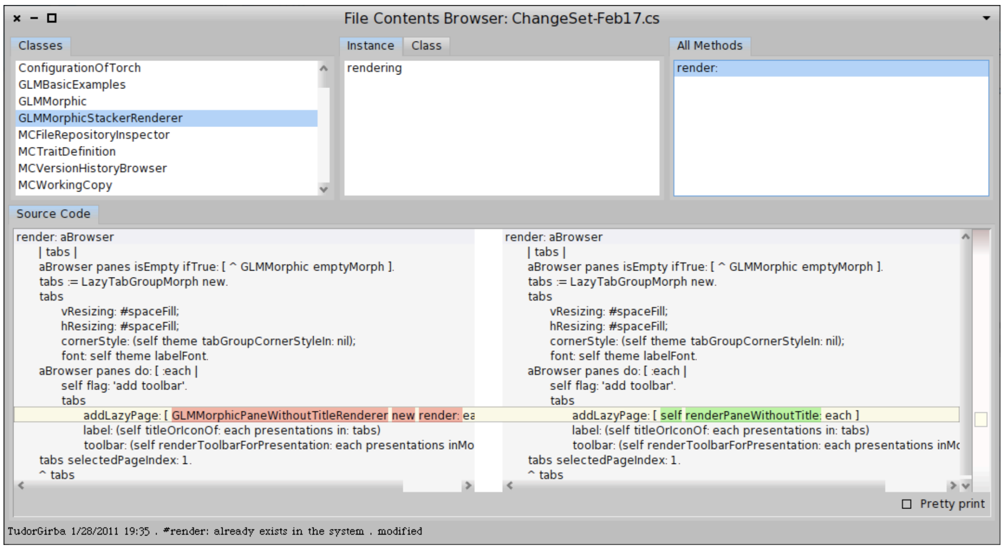


Figure 4.17: *Ring solution for replacing the pseudo classes model in the* FileContentsBrowser.

Our solution for browsing source code file contents using *Ring* is shown in Figure 4.17. The pseudo classes are replaced by *Ring* subclasses. They are shown without background and the *Ring* classes appear with a grey background. Note that the model is larger in comparison to the current pseudo classes, however we deal with all program definitions as first-class entities. Only classes and traits have been extended to provide the specific behavior required by this browser (*i.e.,* loading data from change records/source code files and filing the source code in the image). For this we have extended four classes of the *Ring* model. As classes and traits share the specific behavior mentioned before, we provide the trait RGTFileBasedBehaviorDefinition to avoid code duplication.

We also simplified the original FilePackage and replaced it by RGFileContentsManager. This class is dedicated to read files, to load the source code in the browser, and to offer the possibility to load the code in the image (*file in*).



```
render: aBrowser
    | tabs |
    aBrowser panes isEmpty ifTrue: [ ^ GLMMorphic emptyMorph ].
    tabs := LazyTabGroupMorph new.
    tabs
        vResizing: #spaceFill;
        hResizing: #spaceFill;
        cornerStyle: (self theme tabGroupCornerStyleIn: nil);
        font: self theme labelFont.
    aBrowser panes do: [ :each |
        self flag: 'add toolbar'.
        tabs
            addLazyPage: [ GLMMorphicPaneWithoutTitleRenderer new render: ea
            label: (self titleOrIconOf: each presentations in: tabs)
            toolbar: (self renderToolbarForPresentation: each presentations inMo
    tabs selectedPageIndex: 1.
    ^ tabs
```

```
render: aBrowser
    | tabs |
    aBrowser panes isEmpty ifTrue: [ ^ GLMMorphic emptyMorph ].
    tabs := LazyTabGroupMorph new.
    tabs
        vResizing: #spaceFill;
        hResizing: #spaceFill;
        cornerStyle: (self theme tabGroupCornerStyleIn: nil);
        font: self theme labelFont.
    aBrowser panes do: [ :each |
        self flag: 'add toolbar'.
        tabs
            addLazyPage: [ self renderPaneWithoutTitle: each ]
            label: (self titleOrIconOf: each presentations in: tabs)
            toolbar: (self renderToolbarForPresentation: each presentations inMc
    tabs selectedPageIndex: 1.
    ^ tabs
```

```
TudorGirba 1/28/2011 19:35 . #render: already exists in the system . modified
```

Figure 4.18: *The new FileContentsBrowser.*

Finally, we also included a new FileContentsBrowser shown in Figure 4.18. We did not "port" the tool, but rather we recreated it using the extended *Ring* model and *Glamour*[15], an engine for building dedicated browsers.

4.6.2 Refactoring Browser Source Code Scoping Model

The Refactoring Browser and its current declarative source code scoping model were introduced in Section 4.4.2.1. This case shows how we redesigned its source code model extending three classes of *Ring*.

Figure 4.19 shows our solution for the source scoping model on top of *Ring*. We only needed to subclass three *Ring* classes: RGClassDefinition, RGMetaclassDefinition and RGNamespace to represent classes, metaclasses and namespaces, respectively, with specific behavior of the browser. For methods, we reused 6 of our the 18 methods from the original class and defined a class extension in RGMethodDefinition to add parsing behavior.

The behavior of the Refactoring browser for classes and metaclasses to manage changes, refactoring and conditions was defined in the trait RBTClassDescription to avoid code duplication. An important improvement of our solution is that variables are defined as first-class entities by means of the *Ring* classes, which are not modeled in the current source code scoping model. This change

[15]Glamour: http://www.moosetechnology.org/tools/glamour

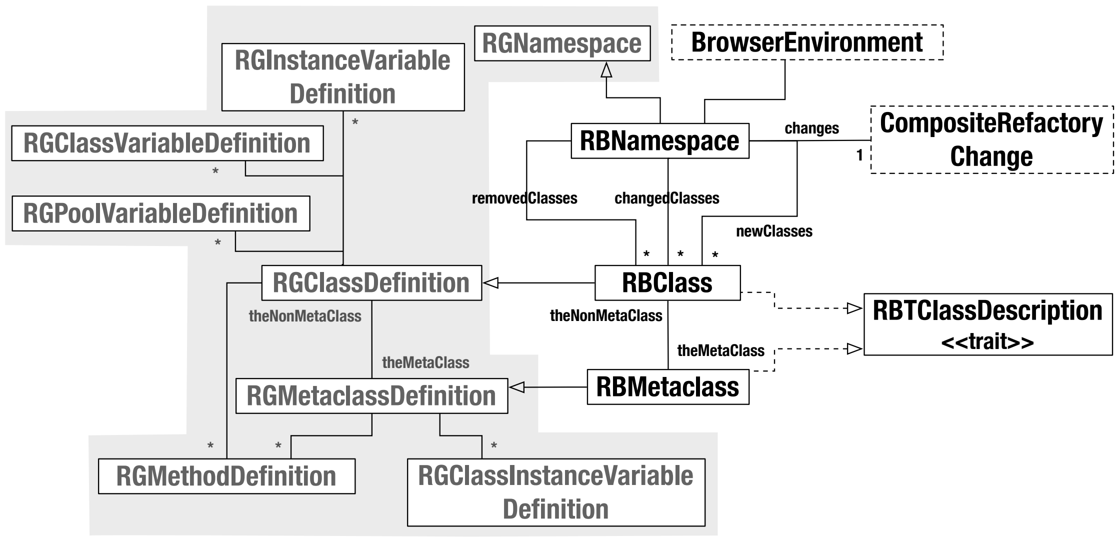


Figure 4.19: *Refactoring Browser new declarative source code scoping model using Ring.*

however did not affect the way the model refers to variables resulting beneficial for our solution as we did not need to modify methods in any model within RB (*i.e., source code model*, *refactoring model* and *changes model*).

Finally, the RBNamespace class that deals with changes of program entities and keeps such changes in separated groups depending on their change status (*i.e.,* new, removed, changed) suffered few adaptations in its API. Concretely, we inherited it from the *Ring* RGNamespace class, removed three methods and one instance variable.

4.7 Discussion

In this section, we discuss three aspects that need to be considered for improving the *Ring* source code meta-model. They do not have an impact on the scientific contributions of this dissertation but refer to technical aspects of our infrastructure that can provide developers with more extensible support for source code representation and the analyses built on top of it.

Applicability to Other Languages. In general, *Ring* can be applied to other object-oriented programming languages even though some of its definitions are specific to Smalltalk (*e.g.,* traits, metaclasses), while some specific constructs of other languages – such as interfaces for Java – are lacking. We could consider to extend *Ring* directly with such definitions or build a layer on top such as a *RingJ* source code model for Java programs. However, the added value of a new layer is not clear, as a language-independent source code meta-model will be very similar to the FAMIX core source model. Building such a language-independent source code meta-model on top of *Ring* would however offer the advantage that the other meta-models, as introduced in later chapters, could be used for other programming languages.

Unifying Models. As shown in Figure 4.12, the *Ring* source code meta-model and the Smalltalk runtime model are independent of each other but they implement a common API. We need to consider whether both models should be merged. The question of knowing whether the runtime entities know their representation is an interesting question from the perspective of a reflective model having another separate and unconnected representation [Ducasse 2009]. The inverse is simpler, keeping track of the runtime representation of entities from the declarative definitions makes it easy and efficient (*e.g.,* Ginsu takes advantage of this). If we cannot simply have either reflective entities that can be disconnected and play the role of declarative ones, merging both models can be an optimal implementation of *Ring*.

Core source model API. We intend to encourage tool reusability by relying on a common API of the main entities (*i.e.,* classes, methods, variables) that basic tools may refer to. This avoids having non-polymorphic APIs for representing entities among different tools. Related to the API the question that arises is: "Are we considering all the definitions that external tools may need?". A typical problem is related to instance variables. Indeed instance variables are not first-class entities in the Smalltalk reflective API even though they are important information for a number of tools. Bridging both worlds and making sure that both structures can be navigated (for example using a visitor) is considered a topic for further investigation.

4.8 Conclusion

In this chapter we presented *Ring*, our source code meta-model to represent the structure of object-oriented software systems. This model is the base of the infrastructure provided in the context of this dissertation as a means to assist integration. *Ring* is a technical contribution that enables us to build a representation for changes and history of software systems and that in turn serve as the underlying models for providing characterization of changes.

First, we introduced the requirements established for modeling the source code of a system taking into account that such model must support history and change representation and analyses.

Second, we presented an analysis of four version control systems data models that we classified in two groups: textual-based and code-based data models. These data models illustrate how version control system store the information and how such information is related to the actual program entities within the source code.

Third, we presented an analysis of seven source code meta-models defined for several purposes. We classified such meta-models in two groups: non-Smalltalk specific and Smalltalk-oriented meta-models. These meta-models served us to determine how current approaches represent program entities and gave us a background for introducing several features into our source code meta-model. In the case of the Monticello source code meta-models, we showed how such models are connected to the data models.

Fourth, we presented *Ring*, our source code meta-model together with its architecture. We illustrated the interaction between *Ring* and other components of our infrastructure. We then showed two usage scenarios were we integrated *Ring* with existing tools from the Pharo as a means to illustrate the use of the meta-model.

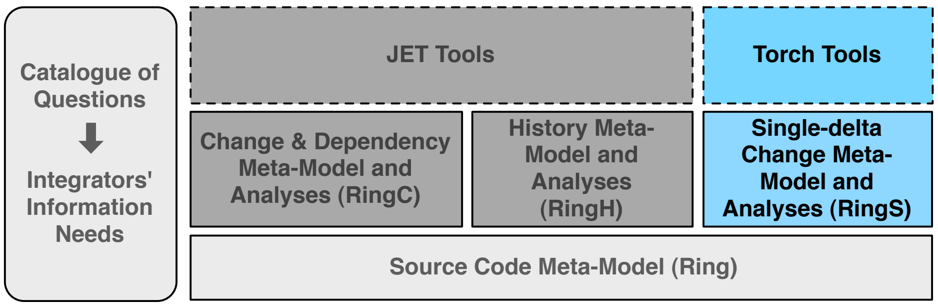
Finally, we discussed three aspects that we should consider as future technical improvements to *Ring*.

CHAPTER 5

Torch: a Dashboard for Grasping Changes

Contents

Contributions Map


Catalogue of Questions
Integrators' Information Needs
JET Tools
Torch Tools
Change & Dependency Meta-Model and Analyses (RingC)
History Meta-Model and Analyses (RingH)
Single-delta Change Meta-Model and Analyses (RingS)
Source Code Meta-Model (Ring)

Overview

This chapter describes our approach for characterizing changes within a single delta and the change meta-model that it is the underlying model for our approach. This approach is an extension of our work published at the *Working Conference on Reverse Engineering* [Uquillas Gómez 2010b]. First, we introduce *Torch*, a characterization of changes and tool support for aiding integrators in understanding the context of single deltas. It provides the integrators' information needs described in Section 3.4 regarding changes within a delta. Second, we describe the architecture of *RingS*, a change meta-model that represents changes between pairs of versions and the context in which these changes exist. It was built on top of the *Ring* source code meta-model presented in the previous chapter. Third, our approach is evaluated by means of usage scenarios, a field evaluation with six integrators and a pre-experimental user study with ten developers. Finally, we compare *Torch* with other approaches that we described in the state-of-the-art in Section 3.5.

5.1 Supporting Change Understanding with Torch

We described the challenges to characterize changes in Chapter 2.3 and we presented the requirements for our solution in Section 2.4. *Torch*[1] is part of our solution to assist integration within a branch. For this, we make use of the integrators' information needs described in Section 3.4 to provide a characterization of changes from a single delta, and offer a means to integrators to support answering questions from the catalogue (presented in Section 3.3.2) regarding these changes.

To support change integration within a branch, and in particular to assist integrators in comprehending changes, our approach, *Torch*, characterizes changes according to structural, authorial and symbolic information. It provides a dashboard displaying object-oriented changes within a delta using class hierarchy inheritance and package distribution visualizations. The visualizations offer an *omnipresent contextual diff* based on a *fly-by help* that allows integrators to explore changes.

Torch characterizes changes from a *RingS* change model. It provides visual tool support to integrators to comprehend changes in context. This means that changes are visualized with respect to a target version. *Torch* helps integrators in making decisions about the integration of changes before performing the actual merging. In addition, it also offers developers a means to understand and control their changes before publishing them.

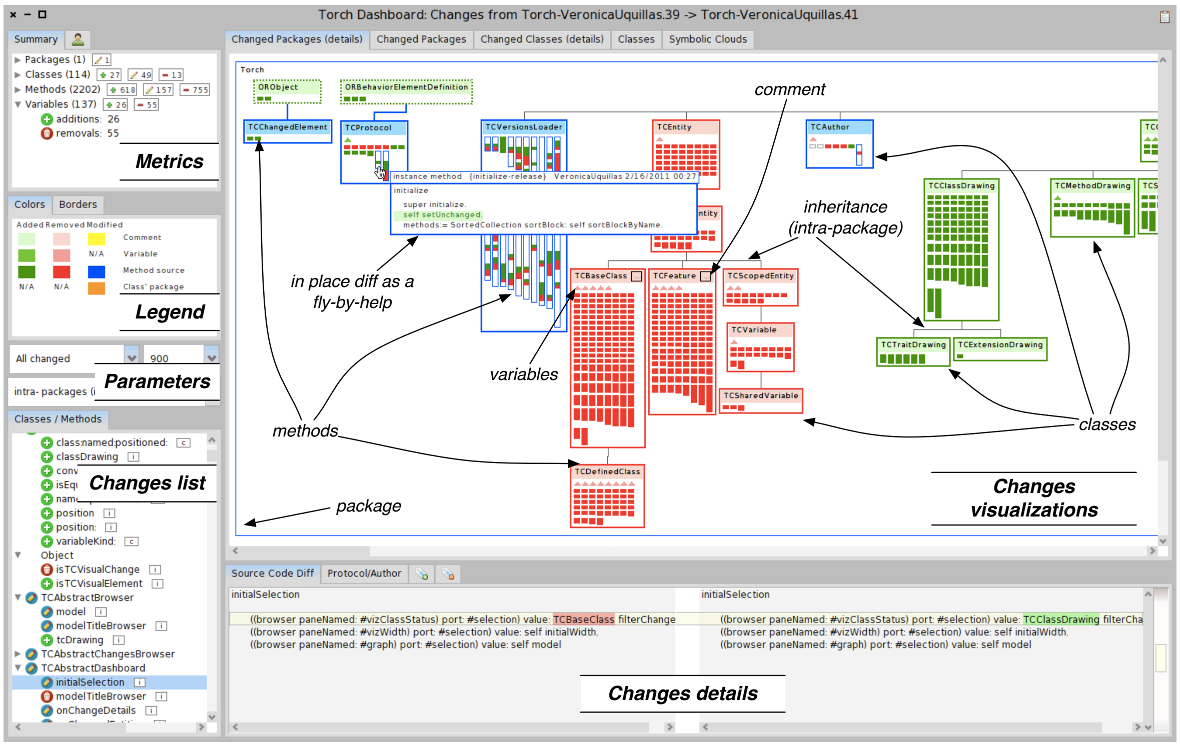
5.2 Layout of Torch

In Figure 5.1 we show the *Torch* dashboard and its main elements. Several visualizations showing the structural representations of changes are the core of the dashboard. It includes a contextual diff as a fly-by help on top of the visualizations to speed up access to the textual information of changes. Torch brings semantic information to changes exploration by combining graphical and textual information. The visual mapping of changes to their structural representation helps users to get a quick overview of the changes and to understand some of their characteristics, such as scope, size, type of change, vocabulary involved, and number of impacted entities. The visualizations can also help integrators to identify patterns among the changes (*e.g.,* feature removals, methods calls replacements), and other aspects such as complexity or semantic impact of the changes. In addition, *Torch* provides a set of metrics about changes per program entity and per author.

In the following, we present an overview of the main elements of the dashboard (as shown in Figure 5.1). In Section 5.3 we present a more extended description of the *Changes visualizations* element of the dashboard.

***Metrics*.** These present the size of the entities impacted by the changes (# packages, classes, methods) as well as measures characterizing the changes themselves (# added, modified, removed entities). The first metric summary shows information per program entity (*i.e.,* packages,

[1]Torch: http://soft.vub.ac.be/torch



```
instance method   {initialize-release}   VeronicaUquillas 2/16/2011 00:27
initialize

    super initialize.
    self setUnchanged.
    methods:= SortedCollection sortBlock: self sortBlockByName.
```

```
initialSelection

    ((browser paneNamed: #vizClassStatus) port: #selection) value: TCBaseClass filterChange
    ((browser paneNamed: #vizWidth) port: #selection) value: self initialWidth.
    ((browser paneNamed: #graph) port: #selection) value: self model
```

```
initialSelection

    ((browser paneNamed: #vizClassStatus) port: #selection) value: TCClassDrawing filterCha
    ((browser paneNamed: #vizWidth) port: #selection) value: self initialWidth.
    ((browser paneNamed: #graph) port: #selection) value: self model
```

Figure 5.1: *Dashboard main elements: the* metrics *give an idea of the size of the changed entities and the actual changes; the* changes list *presents the list of changes and their detailed difference using the* changes details*; the* changes visualizations *present a map of changes structured around packages and classes.*

classes, methods, variables) and per kind of action (*i.e.,* added, modified, removed). The second metric summary presents measures of changes per user and per kind of action, as it may be important to understand who was responsible for the changes.

Legend. Colors are used to represent program entities and kind of actions. They are always visible to help users to get instantaneous information and reinforcement of their knowledge. The legend is the same in the entire dashboard: green for additions, blue for modifications, red for removals, and yellow for modifications of class comments[2]. Icons follow these conventions as well.

Parameters. By default the visualizations display data of *changed* classes and intra-package relationships. Note that the *changed* status refers to added, modified or removed and it is applicable to any program entity. Users can parameterize which classes should display their details by means of the *class status* parameter (*i.e.,* added, modified, removed, unchanged). Inter-package relationships are shown on demand using the *relationships* parameter.

Changes list. Changes representing classes and methods are listed here. Selecting metrics and visual entities filters changes from this list.

Changes details. Class definitions and comments, method source code, authors, protocols, and symbols (*i.e.,* vocabulary involved) are mainly presented using a diff view in this element of

[2]Modifications of class comments appear in yellow to increase the contrast with the background.

the dashboard.

Changes visualizations. Corresponds to the main element of the dashboard that visually shows unchanged and changed program entities with their structural representations (*e.g.,* see Figure 5.1). The changes are highlighted respecting the *Conventions*. The visualizations include a contextual fly-by help that supports an in-place diff view.

5.3 Dashboard Visualizations

The comparison of two versions (*i.e.,* base and target version) is graphically presented in the main element of the dashboard, named *Changes visualizations*. This element shows software visualizations with program entities, their relationships, and the vocabulary involved in changes (*i.e.,* keywords from the source code). Additionally, *Torch* does not only show changed entities between pairs of versions but also unchanged entities within the *target* version, providing a complete visual, structural representation of the target version with the context and characteristics of changes.

Software visualization. is the use of any graphic means (typography, graphic design, animation, etc.) to facilitate human understanding of complex software systems [Stasko 1998]. A well-conceived visualization [Tufte 2001, Ware 2004] triggers the human brain's inherent capacity to combine complex information from visual clues, making it possible to quickly understand a complex software system. Furthermore, visualizations allow users to *pre-attentively process*[3] the visual information: rather than having to search for specific information (*e.g.,* by extracting it from the source code), visualizations can immediately draw a user's attention to specific parts of a system.

Philosophy behind Torch's visualizations

- Do not restrict the level of detail of the information provided
- Provide a single convention for multiple visualizations
- Maintain the link between a graphical program entity and its source code
- Maintain the link between the different visual representations of program entities

In object-oriented programs two main definitions are available for structuring a system: the package containment[4] and the class inheritance relationships. In particular, it is important to understand a change in its context since changes made in a class will impact subclasses or lead to the "yoyo effect" [Taenzer 1989, Wilde 1992]. Even a list of changes does not offer such a context and an overview of the changes at the same time. This is why we design visualizations structured around these two main axes: packages and inheritance hierarchies.

Before describing the main visualizations, in the rest of this section we explain the visual representation of entities and the fly-by help utility.

[3]Pre-attentive processing is the unconscious accumulation of information from the environment [Van der Heijden 1996]. First, available information is pre-attentively processed. Then, the brain filters what is important for further and more complete analysis by conscious (attentive) processing [Van der Heijden 1996].

[4]Note that package containment is not limited to object-oriented programs.

5.3.1 Entities Representation

Torch uses two shapes for representing program entities: rectangles and triangles. Rectangles represent packages, classes, traits and methods; triangles represent attributes. The *Torch* dashboard uses the same representations and conventions for displaying classes and traits, with the exception of how the border is visualized. Dashed borders are used for traits and class extensions. Three kind of edges are used for representing relationships: (a) arrowed edges for class-inherits-class, (b) dashed arrowed edges for class-uses-trait, and dashed edges for class-is-extended-in-package. Colors are mapped onto a kind of action (*i.e.,* added, removed, modified, moved) of a program entity or inheritance relationship.

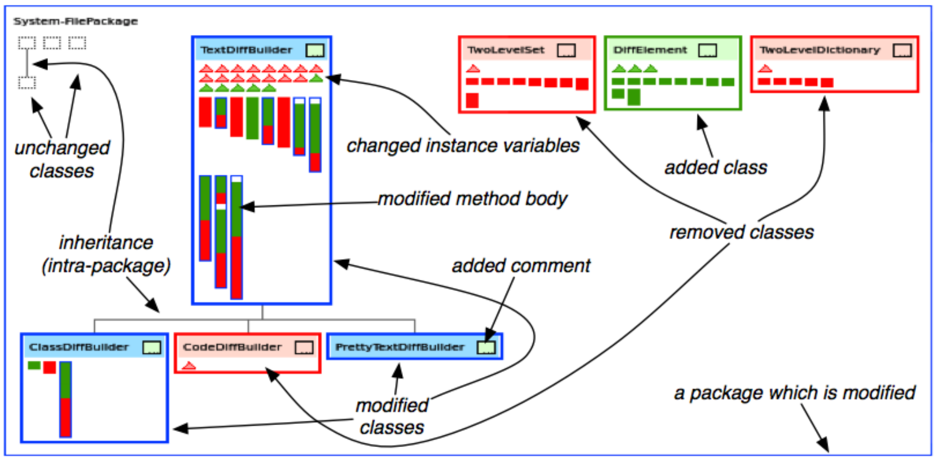


Figure 5.2: *Package containing unchanged classes (small dashed grey rectangles), removed classes (red rectangles), added classes (green rectangle) and modified classes (blue rectangles). Classes contain attributes (triangles) and methods (bars).*

Packages. Figure 5.2 shows the modified System-FilePackage and its changed classes using a structural representation of classes. A package is displayed as a large rectangle containing all its classes and traits (not only changed ones). Inside, when possible, classes are organized in class hierarchies, and they show changes using any of the class representations explained later. Unchanged classes and traits are represented by small dashed boxes.

Classes. A class has two visual representations for its changes: *structural representation* and *condensed representation*. Figure 5.3 shows both class representations making use of three classes (added, removed and modified class). Note that the color of the border of a class and a light version of the same color as the background of the class name represent whether the class was entirely added (green), removed (red) or simply modified (blue). If the class comment of a class changed, this is indicated with a colored box next to the class' name (*i.e.,* the comment was added - green, removed - red or modified - yellow).

- *Structural representation.* A class is displayed using sections: the class name section, attributes section and methods section (see classes on the top of Figure 5.3). DiffElement and ScreenController have changed attributes, methods and comment, whereas CrLfFileStream has changed

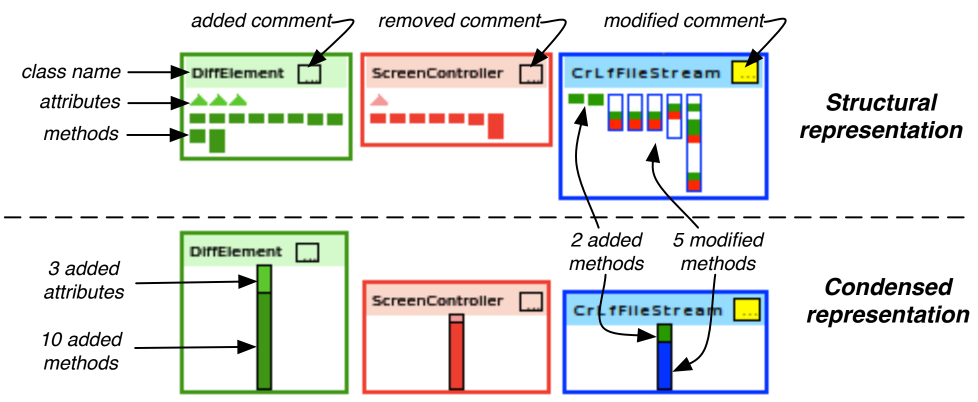


Figure 5.3: *Structural and condensed visual representation of classes*

methods and comment, therefore the attributes section is hidden. The height of the bar representing a method is related to its number of lines of code. Modified methods have a blue border and may include three inner colors which are mapped to the changes per line in their source code (added line – green, removed line – red, and unchanged line – white), *e.g.,* 5 methods in the class CrLfFileStream.

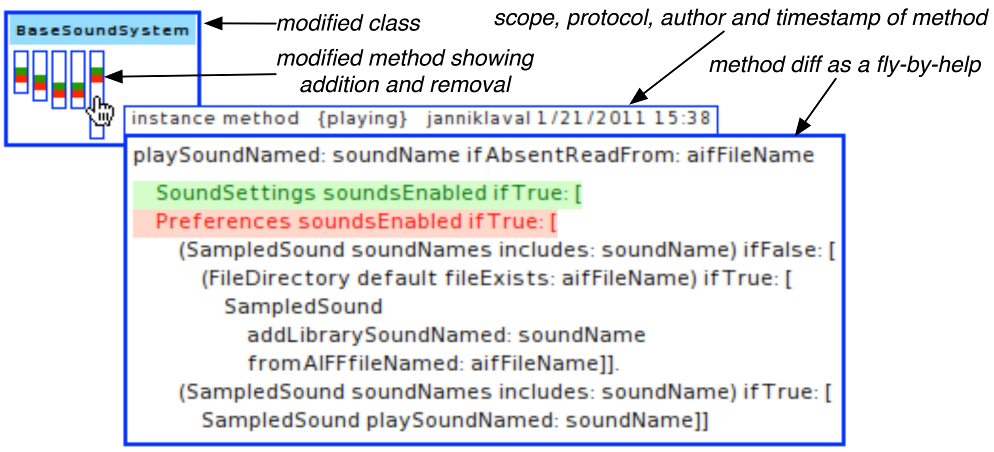
- *Condensed representation*. Changed attributes and methods may also be presented together as a single bar summarizing the number of changes (see classes at the bottom of Figure 5.3). The bar is composed of colored segments. Each segment groups changes (*e.g.,* added attributes, removed methods), uses a color for that group of changes (*e.g.,* added methods in dark green, added attributes in light green, modified methods in blue) and has a height (the number of those changes). This visual representation also includes a class name section as the *Structural* representation.

5.3.2 Fly-by Help

Within our visualizations we provide two **omnipresent** fly-by helps to show the source code of any method and the structure of any class or trait. They are available at any time when the user hovers over a method, class or trait.

Diff as a fly-by help. The main visualization of the dashboard shows the structural representation of changed classes/traits and makes use of a fly-by help to show the source code differences (diff) and other information of any method. One important design point is that most of the visual representations can be hovered over to display the associated code without having to change tool/pane.

Figure 5.4 shows a source diff as a fly-by help. It shows a method's code and highlights line additions in green and removals in red. The background color of added and removed lines appears in light green and light red respectively. This allows us to show empty lines that were added to or removed from the code. In addition, extra information of a method is displayed on top of the source diff: the scope (*i.e.,* instance or class method), the protocol, the author and the timestamp when the change happened.


BaseSoundSystem
modified class
modified method showing addition and removal
scope, protocol, author and timestamp of method
method diff as a fly-by-help

```
instance method  {playing}  janniklaval 1/21/2011 15:38
playSoundNamed: soundName ifAbsentReadFrom: aifFileName
  SoundSettings soundsEnabled ifTrue: [
  Preferences soundsEnabled ifTrue: [
    (SampledSound soundNames includes: soundName) ifFalse: [
      (FileDirectory default fileExists: aifFileName) ifTrue: [
        SampledSound
          addLibrarySoundNamed: soundName
          fromAIFFfileNamed: aifFileName]].
    (SampledSound soundNames includes: soundName) ifTrue: [
      SampledSound playSoundNamed: soundName]]
```

Figure 5.4: *Omnipresent code browsing: diff as a fly-by help.*

Full class structure as a fly-by help. Most of our visualizations that display classes only include *changed* attributes and methods. *Torch* complements this information by also offering a fly-by help of the *full* class structure that appears when hovering over a class name, shown in Figure 5.5 (right). Developers can see unchanged attributes and methods that are defined in a class (*i.e.,* white bars and triangles with grey border), and thus have a real idea of the number of changes that affected that class. Furthermore, the fly-by help is also available for unchanged classes, allowing developers to observe the structure of any class in the dashboard.

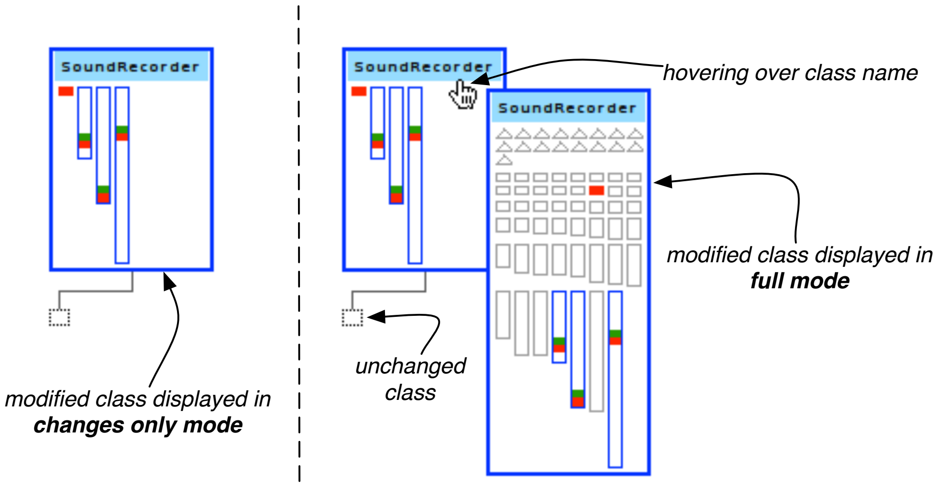


Figure 5.5: *Class displayed in changes only mode (left). Omnipresent class structure: class displayed in full mode as a fly-by help (right).*

5.3.3 Package-centric Visualizations

Package-centric visualizations provide the structural context of any existing change, by distributing classes and traits in packages and methods in classes or traits. Three visualizations are proposed and represent the most complete source of information that Torch offers to integrators. Each has a special purpose for supporting the understanding of changes. Figure 5.11 shows an example usage of this kind of visualization.

- *Changed Packages (details).* When comparing versions with many unchanged packages, we decrease the size and complexity of the visualizations by only presenting changed packages. The purpose is to provide an integrator with a visual, structural representation of *changed* entities. Each package shows its classes and the inheritance relations defined within that package. Each changed class shows its structural definition only containing changed methods and attributes, allowing an integrator to focus only on what was changed in that class.

- *Changed Packages (condensed).* This visualization only presents changed packages. Its purpose is to further minimize the visualization of the changes by using the condensed representation of changed classes.

- *Packages (condensed).* This visualization differs from the previous ones by also presenting unchanged packages. Classes are shown with the condensed representation. The goal is to show the general impact of changes (location, size and complexity) over the whole version. An integrator can compare the size of changed versus unchanged packages and can observe and explore classes defined in unchanged packages that may have relationships with changed classes (*e.g.,* inheritance).

5.3.4 Class-centric Visualizations

Class-centric visualizations exclude packages. This means that classes and traits are linked together by their class inheritance and trait-use relationships. We also refer to this kind of visualizations as inheritance-based. They are included in the dashboard because we consider that omitting package containment relationships can provide an integrator with different views of how all the classes under analysis are related. An example usage of this kind of visualization is shown in Figure 5.15.

- *Changed Classes (details).* This visualization shows the changed classes and other classes that are part of its class hierarchy (*i.e.,* superclasses and subclasses). The purpose of this visualization is to show a more complete hierarchy of the changed classes allowing an integrator to identify which unchanged classes may be affected by external changes.

- *Classes (condensed).* This visualization shows the same information as the *Packages (condensed)* visualization. However, it provides the general impact of changes by displaying all classes and their inheritance relationships without package containment relationships. Classes are shown using the condensed representation to minimize the size of the visualization.

5.3.5 Symbolic Clouds

Symbolic Clouds are the third kind of visualization presented in the dashboard. They show the vocabulary that changed instead of changed program entities. The goal of symbolic clouds is to give hints of the developers' intentions while changing the source code (*e.g.,* whether the change vocabulary is different from the one of the application or new vocabulary is introduced).

The clouds are built by extracting method invocations, class references, and accesses to instance variables and to three literals values (*i.e.,* nil, true, false) from changed source code. Each symbol is associated with the number of its occurrences in the source code and with a color defined in the

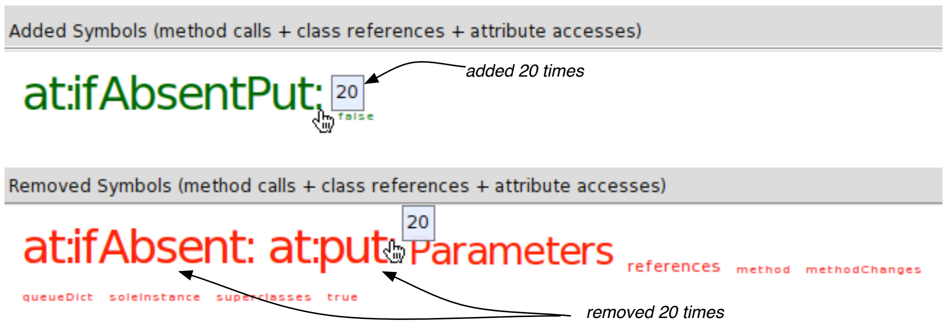


Figure 5.6: *Added and removed symbolic clouds.*

conventions (*i.e.,* green for added symbols and red for removed symbols). The number is mapped onto a font size that is used for drawing that symbol.

Three symbolic clouds convey the added, removed and mixed symbolic information. Figure 5.6 shows the two first clouds applied to the scenario where the combined method calls at:ifAbsent: and at:put: (red symbols) were replaced by the method call at:ifAbsentPut: (green symbol). Figure 5.20 shows another example usage based on the mixed symbolic cloud.

5.4 Supporting the Answering of Integrator Questions

In this Section, we discuss how *Torch* can aid in answering the questions that integrators ask themselves when performing integration activities. We introduced a catalogue of 64 questions in Section 3.3 that served to identify the integrators' information needs in order to assist them during the integration process. Multiple questions motivated several of the features provided by the dashboard such as the diff as a fly-by help, symbolic clouds, metrics of changes, package-centric and class-centric visualizations. These features ease answering several questions.

We guide our discussion based on the 5 categories used to classify the questions: (a) author/owner questions, (b) behavioral questions, (c) structural change characterization questions, (d) infrastructure questions, and (e) temporal and change stream questions.

Author questions. Four of the six questions in this group can be fully answered with *Torch* (*e.g.,* "Who made this change?" or "Who wrote the original code that was changed?"), and the question "What is the general quality of the change committer?" can be partially answered. The author of any change and the committers of the versions compared are shown in the change lists. The diff as a fly-by help also provides this information for any change on the visualizations. Answering "How many people have contributed to this sequence of changes?" is not possible with *Torch* because it only visualizes a delta and not a stream of changes.

Behavioral questions. Six out of the fourteen questions in this group can be partially supported (*e.g.,* "What is the reason for this change?" or "What are the implications of this change for API clients?"). For this, the integrator can use the descriptive and structural information provided by the dashboard. Questions such as "Does this change improve the quality?" or "Is this change correct?" are not supported by *Torch*, and in fact both questions are not supported by our whole approach because their subjectivity. The questions related to test coverage such as "Is the change covered by tests? What is the coverage?", "Did this change fix/break tests? Which tests?" or "Did the tests work before the changes?" are not supported because currently *Torch* does not take tests into account.

Structural change characterization questions. *Torch* provides answers to 10 out of the 19 questions in this group (*e.g.,* "What is the scope of this change? (which/how many classes/packages/..., is local/global?)" or "Is this change confined to a single package?"). They are related to change size, change scope and change structure. Two of them are partially supported (*i.e.,* "Does the change follow rule checking/conventions?" and "Is the vocabulary used in the change consistent with the one of the system?") by providing the symbolic clouds.

Questions such as "What are the required structural dependencies?" or "What other changes depend on this change?" are not supported because *Torch* does not analyze prior changes that may be needed by the changes within the delta that is visualized in the dashboard.

Infrastructure questions. The question "To which bug entry does this change relate?" regarding the bug tracking infrastructure can be partially supported by *Torch* when the commit logs include that information. *Torch* does not take other artifacts like documentation or bug reports into account, and therefore such questions lie outside the scope of our approach.

Temporal and change stream questions. Most of the questions in this category are not supported by *Torch* because they are related to changes within a stream (*i.e.,* sequence of successive versions). *Torch* partially supports 2 questions out of the 23 (*i.e.,* "When was this change made?" and "What else changed when this code was introduced or changed?") by providing the timestamp of the changes and the visualized structural information.

5.5 RingS: a Single-Delta Change Model

In Chapter 4 we presented the *Ring* source code meta-model. In this section, we present *RingS*, our change meta-model for representing changes within a single delta (*i.e.,* a set of changes between two versions) extending our source code model. *RingS* enables the characterization of changes provided by *Torch* to assist integration within a branch. As *Torch* requires us to know both which entities changed, and in which way they changed, we introduce *RingS* to represent descriptive and structural information. Both kinds of information are part of the integrators's information needs described in Section 3.4. By means of this information we can support answering questions from the catalogue presented in Section 3.3.2 from within *Torch* regarding changes within a delta.

5.5.1 Architecture of RingS

RingS is a change meta-model built on top of *Ring*, and it is the underlying model for version comparisons and characterization of changes within a single delta. With *RingS* we can compare a pair of versions (*i.e.,* base and target), or multiple pairs of versions at the same time. Since our tools target the Pharo environment and the Monticello version control system, this comparison takes into account that a commit may consist of multiple package versions published separately, therefore a delta may be the result of comparing pairs of package versions.

RingS models the changes between a base and target version by providing a representation of the entities in the target version, along with the respective status for each entity (*e.g.,* added, unchanged). Moreover, any entity is aware of which of its properties changed (*e.g.,* for a method its source code may have changed). To provide the context of the changes, we are required to model the whole target version and not only the differences between the base and the target versions.

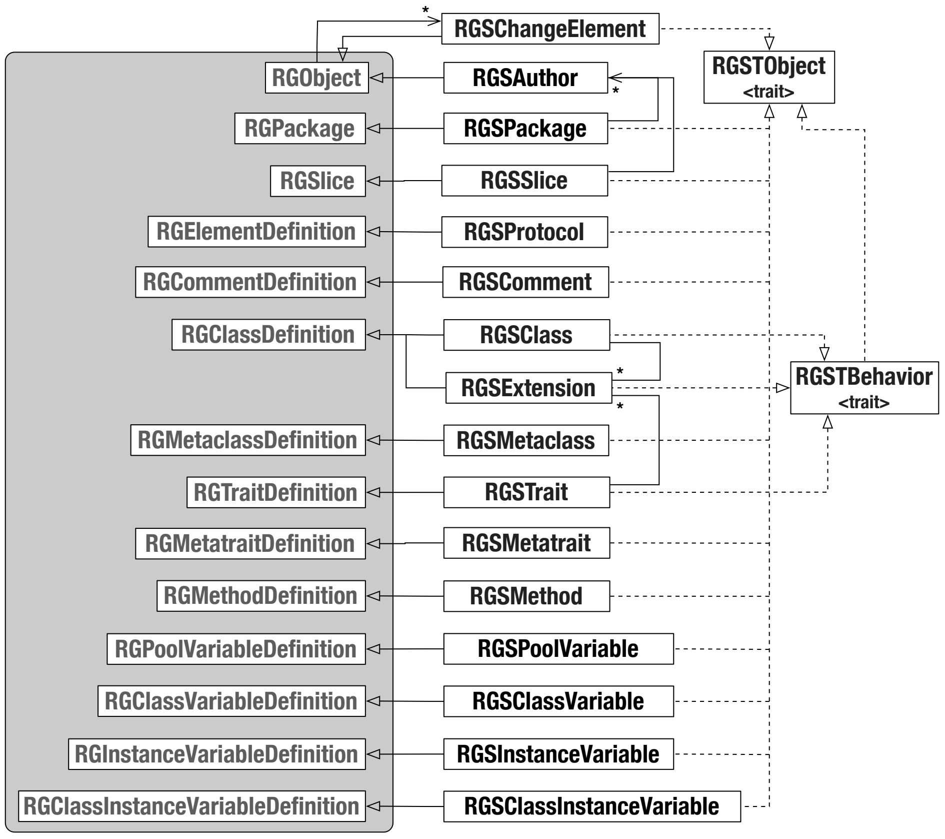


Figure 5.7: *RingS single-delta change meta-model - Key classes.*

In Figure 5.7 we show the *RingS* single-delta change meta-model (classes with white background) and how it extends the *Ring* source meta-model (classes with grey background). Note that in this diagram we mostly show the inheritance relationships between *RingS* and *Ring* classes. Other relationships such as associations between *RingS* classes are the same as in the *Ring* meta-model shown in Figure 4.13, *e.g.,* a RGSClass object may have zero or more RGSInstanceVariable objects.

Base Classes

To manage the status of every object and the status of its properties, we defined a core trait, namely **RGSTObject.** It is used by the classes in the model, as seen in the figure (right).

RGSTObject defines the behavior for setting and retrieving the status of an object, such as *added*, *removed*, *modified* and *unchanged*. It also manages the attributes (*i.e.,* variables in Smalltalk) of any object when such attributes are changed.

The changes between versions may be represented as changed properties within program entities (*e.g.,* the superclass of a class, the source code of a method, etc). To deal with changed properties in *RingS*, the class **RGSChangedElement** has been defined. This class also uses the RGSTObject trait, and allows RGSChangedElement objects to be marked as *added*, *modified* or *removed*. For example, suppose we compare two versions in which the superclass of class Monkey was changed from the class Animal to the class Mammal. In a *RingS* model the status of the Monkey object is *modified*, and its *superclass* instance variable has a RGSChangedElement object with the Animal and Mammal objects corresponding to the *previous* and *current* values respectively.

Another trait – RGSTBehavior – is defined for sharing behavior between classes and traits that manage methods, protocols and variables. This trait uses RGSTObject and therefore it provides classes and traits with the RGSTObject behavior as well.

We group all entities from the target version(s) in a so-called *slice*, which is a concept that comes from Monticello. A slice is a set that groups entities from one or multiple packages, *i.e.,* it contains packages, classes and methods. A slice is used to version multiple packages together and to alleviate partially the lack of explicit commits. *RingS* models slices with the class **RGSSlice.**

Additional Classes

RingS defines two additional classes with respect to *Ring* for managing the author of the changes, and protocols of methods and class extensions.

Authors are defined as first-class entities by means of the class **RGSAuthor.** This allows us to associate metrics to each author (*e.g.,* number of additions). An author can be directly associated with a class and a method (independently of the status of such entity). The authors of a slice are defined as the set of authors of all the entities contained within the slice.

In Smalltalk methods can be annotated with the category to which they belong, known as a protocol. It serves as a means to organize methods by their underlying purpose. Therefore, a protocol does not affect the semantics of the system. However, instead of defining protocols as String objects, in *RingS* protocols are defined as first-class entities. They are modeled with the class **RGSProtocol** that is a subclass of RGElementDefinition from *Ring*. Therefore a protocol is also an element of a class or trait.

Smalltalk supports class extensions [Bergel 2005], *i.e.,* developers are able to add methods to classes that are contained in packages different from the ones which the classes belong to. For example, if class C is contained in package P and method foo of class C is contained in package R, then method foo is a *class extension* of C. This is a simple mechanism that allows developers to add behavior to existing entities without subclassing them in other packages.

In *RingS*, class extensions are also represented as first-class entities by means of the class **RGSExtension.** This class is a subclass of RGSClass. Note on the figure that a RGSClass object or a RGSTrait object that models classes and traits may contain multiple class extensions.

Graphic Layer

Another characteristic of the program entities defined in the *RingS* change meta-model is that each is associated to a class that defines how such entity is drawn on the dashboard. By means of this, we can easily reuse the graphic representation of entities throughout *Torch*.

The graphical logic of several program entities is shown in Figure 5.8. For each entity that needs to be drawn in the dashboard a class has been defined (*i.e.,* slice, package, class, trait, extension, method, variable and protocol). We call them *graphic classes*. They inherit from the class RGSGraphicObject.

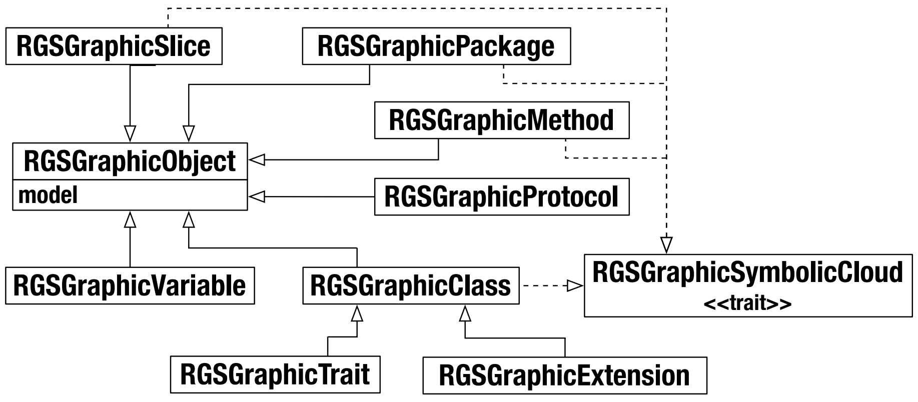


Figure 5.8: *RingS single-delta change meta-model – Graphic classes.*

This class knows the model (*i.e., entity*) to be drawn and how to draw it. Several objects (*e.g.,* methods) are able to display the symbolic clouds explained in Section 5.3.5. They use the behavior defined in the trait RGSGraphicSymbolicCloud.

5.6 Torch Usage Scenarios

In this section, we show as part of an evaluation of our approach how *Torch* characterizes changes within a single delta by using the integrators' information needs (described in Section 3.4). More precisely, we illustrate the usage of *Torch* by applying it to the changes[5] of the Pharo project. This is a qualitative evaluation performed by the author of this dissertation. We took the repositories located at www.squeaksource.com/PharoInbox and ss3.gemstone.com/ss/PharoInbox containing the commits of the Pharo versions 1.3 and 1.4, and the repositories located at www.squeaksource.com/PharoTreatedInbox and ss3.gemstone.com/ss/PharoTreatedInbox containing the commits once they have been integrated into the current release. By means of this internal evaluation, we show how *Torch* helps understanding and characterizing some typical scenarios. Note that *Torch* can be applied to any other change scenario. The purpose of this section is to give an idea of how the dashboard reflects the changes.

Torch can supports integrators answering several questions from the catalogue presented in Section 3.3.2. Especially, the ones in the authorship/ownership and structural changes categories. Such as: "Who made this change?", "What kind of change is it? (Bugfix/New feature/Refactoring/Documentation)", "How large is the change?", "What is the scope of this change? (which/how many classes/packages/..., is local/global?)", etc.

In the following, we apply *Torch* to six scenarios and discuss how it assists integrators in obtaining their information needs.

5.6.1 Removing a feature

In particular cases, certain features that became irrelevant or unused are removed. An integrator can easily detect a feature removal with the *Torch* dashboard. The pattern is simple (*i.e.,* mostly removed

[5]Our usage scenarios are based on changes that were published as slices. The name of a slice has the following notation: `"SLICE"-("Issue"-numberOfIssue)-description-committer.numberOfVersion`

entities which appear all in red) but it can be subtle: indeed clients may need to remove references to the removed features (*i.e.,* such clients can be represented as modified classes and modified or removed methods). The dashboard provides a broader view than a list of changes. It shows the magnitude and impact of such a removal on the system using the structure of its program entities.

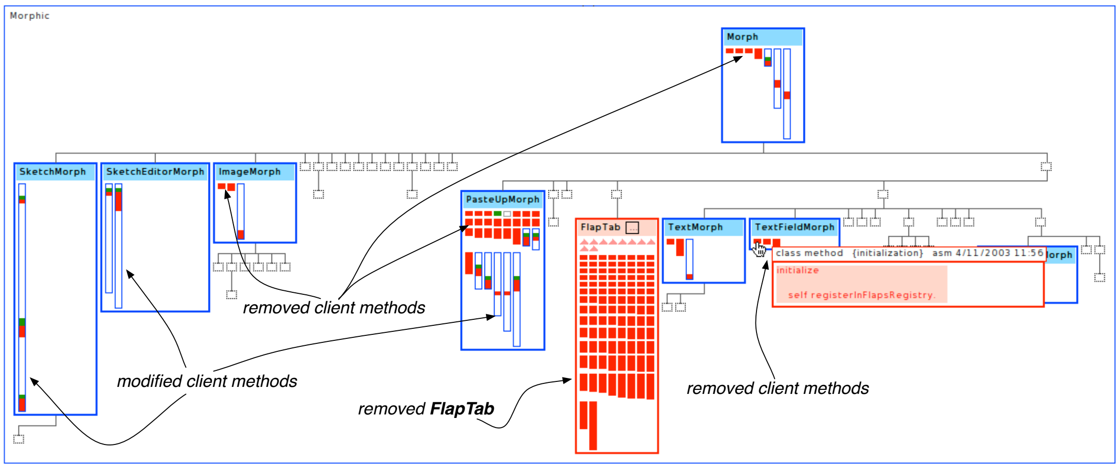


Figure 5.9: *Removing the feature* **FlapTab** *: several methods in clients were* modified *and other methods were simply* removed – *SLICE-FlapRemoval-AlainPlantec.1 (Oct. 17th 2009).*

Figure 5.9 shows the removal of the UI feature *FlapTab*. The class FlapTab was completely removed (all its methods are red and the class border is red as well indicating that the class has been removed), as well as many of its client methods. We can also see that some client methods got adapted (*i.e.,* modified) by removing a few lines of code (methods with blue border).

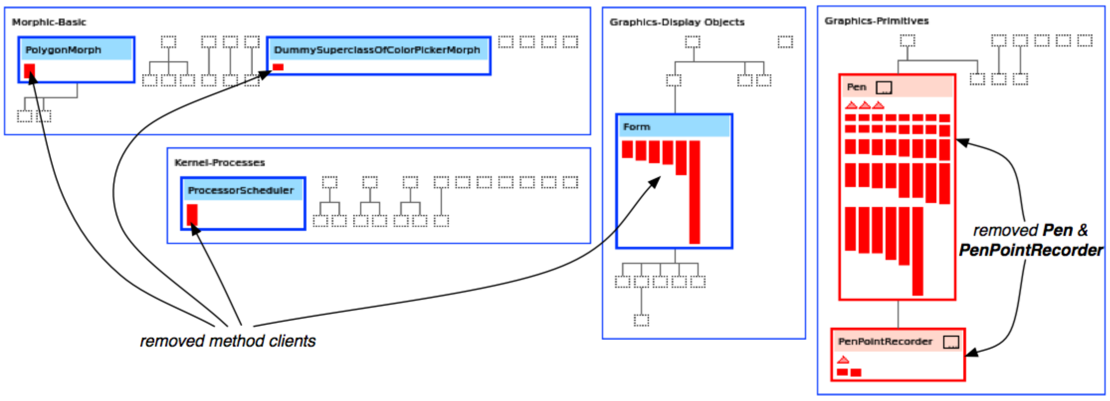


Figure 5.10: *Removing the feature Pen: classes* **Pen** *and* **PenPointRecorded** *were removed and their client classes also removed entire methods – SLICE-2163-RemovePenAndPenPointRecorder-MarianoMartinezPeck.2 (March 18th 2010).*

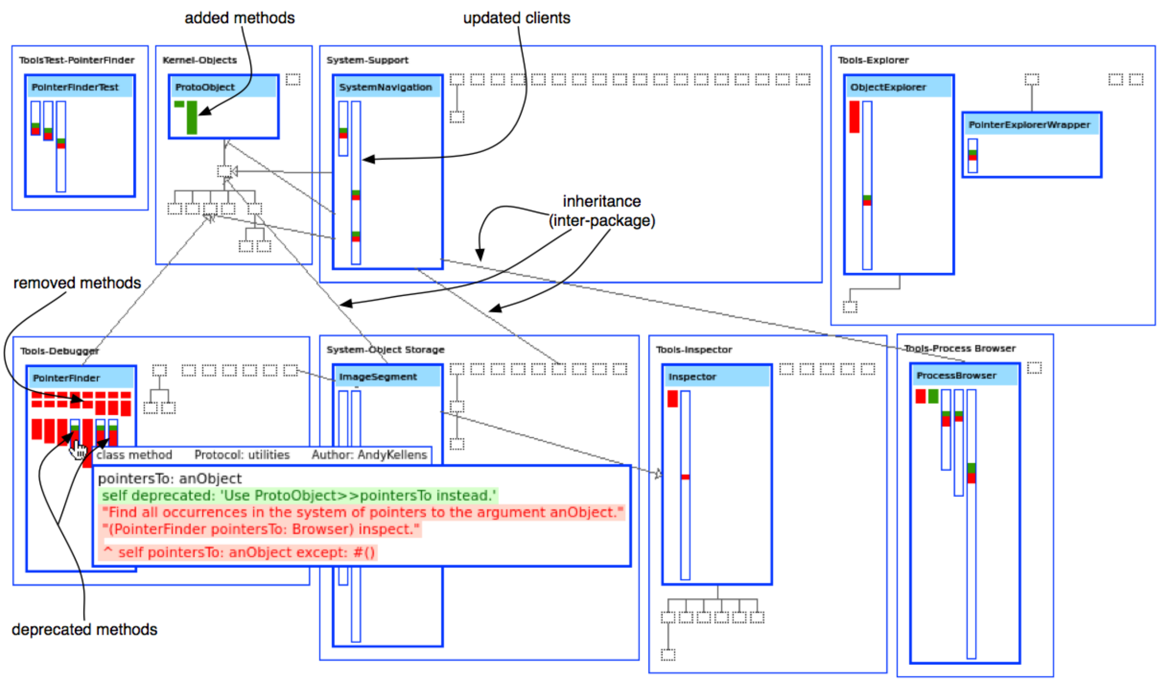
Figure 5.10 shows the removal of the feature *Pen*. The classes Pen and PenPointRecorder were completely removed as well as their client methods.

Both cases show how *Torch* visually eases the identification of feature removals. Regarding the integrators' information needs, the *kinds of actions* play a key role (in these cases removals are predominant while modifications may appear for client adaptations). An integrator can also observe the different structural information of changes such as the *kinds of entities* that are affected (*e.g.,* methods, classes, variables, packages), the *structure* (*e.g.,* class hierarchies within packages, variables within a class), and the *scope* of the changes (*e.g.,* the first case shows the removal of FlapTab within a single package, while the second case shows the removal of Pen cross-cutting 4 packages). At the same time, descriptive information such as the *size* of the changes is provided, and therefore an integrator can answer questions such as "How large is the change?" by only observing the number of affected entities in the dashboard. The diff as a fly-by help can ease answering *author/owner* questions.

5.6.2 Removing a feature and deprecating its API

Changes associated to a feature removal are mostly deletions of source code. However, the complete removal of a feature is not often a practice that is adequate and deprecating the API is an important action to help clients adapt to the new interface. In addition it may happen that the feature is kept while the objects responsible to implement it are changed.

In this section we show the pattern of feature deprecation. This pattern is a variant of the feature removal pattern discussed above. The most important difference is that in this pattern, facilities are introduced to make developers aware that they are using deprecated code.



```
class method     Protocol: utilities     Author: AndyKellens
pointersTo: anObject
 self deprecated: 'Use ProtoObject>>pointersTo instead.'
 "Find all occurrences in the system of pointers to the argument anObject."
 "(PointerFinder pointersTo: Browser) inspect."

 ^ self pointersTo: anObject except: #()
```

Figure 5.11: *Removing the feature* **PointerFinder** *and deprecating its API: its functionality was substituted by another tool –* `SLICE-PointerFinderRemove-AndyKellens.1` *(June 11th 2010).*

This case shows that in Pharo existed two tools to identify memory leaks (trace pointers), namely PointerFinder and PointerExplorer. The developers opted to remove this duplicated functionality by deprecating PointerFinder.

Figure 5.11 shows the effect of the feature deprecation. Nearly all the methods of the class PointerFinder were removed as shown by the red methods, and three methods were modified (*i.e.,* marked as deprecated) as shown by the green and red stripes within the bars with a blue border. The source code of the method pointersTo: before and after the deprecation is shown in the omnipresent diff as a fly-by help.

To ease migration of existing client code of PointerFinder, the developers added a couple of methods offering access to the pointer tracing functionality in ProtoObject (see class at the top, second package). All other changes (mostly modifications in methods) correspond to the clients of PointerFinder that now make use of the new implementation.

Note that this case shows a different visual pattern than when fully removing a feature. As can be seen in Figure 5.11, no class removals exist. Instead multiple method removals within a single class are present. The integrator can easily identify this pattern but making use of the diff as a fly-by help on the affected program entities and the *structural information* provided. For example, an integrator can first focus on the class with the majority of removed methods (in this case PointerFinder), and second inspect the changes cross-cutting other packages to verify whether they correspond to users of the deprecated feature. By exploring the changes in the dashboard the integrator can answer questions such as “Do all the changes within the commit belong together or are they unrelated?”

5.6.3 Introducing a feature

The dashboard reflects new features as a set of added classes and/or methods, along with some modifications in existing classes (*i.e.,* method modifications) that make use of the new features. When a feature introduction is submitted as a single set of changes, it is easily identified in the dashboard. Otherwise having other unrelated changes in the same delta may decrease the visibility of this pattern.

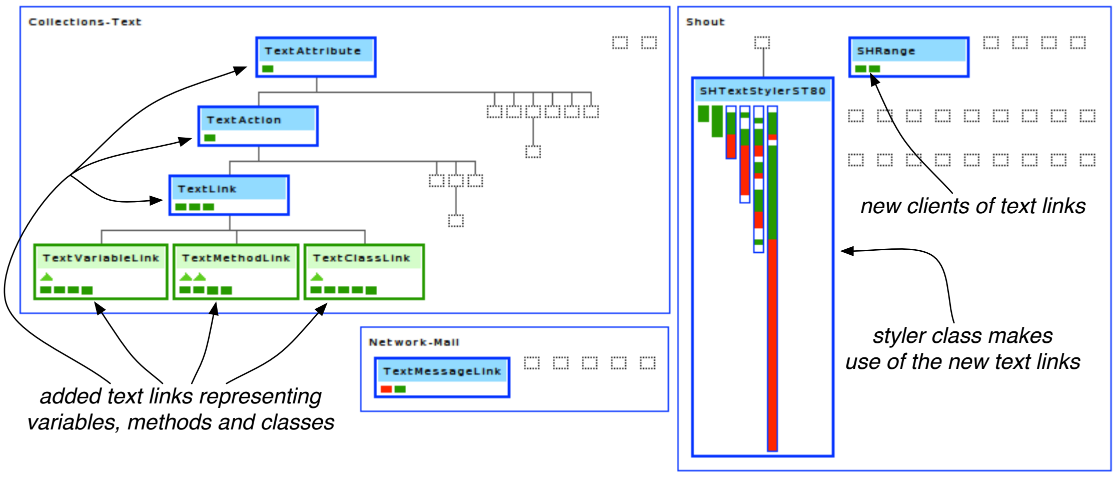


Figure 5.12: *Introducing features: new variations of text links for code styler – `SLICE-Issue-5233-Support-Semantic-Source-Links-CamilloBruni.5` (Feb. 6th 2012).*

Figure 5.12 shows the introduction of three variations of text hyperlinks. They are TextClassLink, TextMethodLink and TextVariableLink hyperlinks. The code styler feature SHTextStylerST80 (large class in the package on the right) is the main user of these hyperlinks. The boxes of the three added classes have a green border to represent additions. We see these classes as indirect subclasses of the TextAttribute root class, accompanied by few method additions in several classes in that hierarchy. Browsing the code of other changed classes with the fly-by help confirms that they are clients of the new features, in particular the styler SHTextStylerST80, TextEditor and Paragraph (not displayed in the figure).

For an integrator, detecting the addition of a feature on the dashboard is very similar to a removal. The only difference is that instead of having multiple *red* classes or methods, with a new feature the dashboard shows multiple *green* classes and methods. This is combined with the *blue* classes representing the clients using the new feature (*i.e.,* classes that added, modified or removed methods). Note again that *structural* information of the changes such as kinds of actions, kinds of entities, structure and scope allow an integrator answering questions such as "Does this change define only one feature?" or "What is the complexity of the changes/of the touched entities?".

5.6.4 Pushing up methods / Introducing methods in a class hierarchy

Since classes are structured in inheritance trees and methods may impact multiple classes, it is important to understand where the changes happen in an inheritance tree. *Torch* provides package-centric and class-centric visualizations that both make use of the inheritance relationships between classes. However, when changes affect an inheritance tree that cross-cuts several packages, the class-centric visualizations show a better view of these changes and ease the identification of change patterns (*e.g.,* a push up method refactoring).

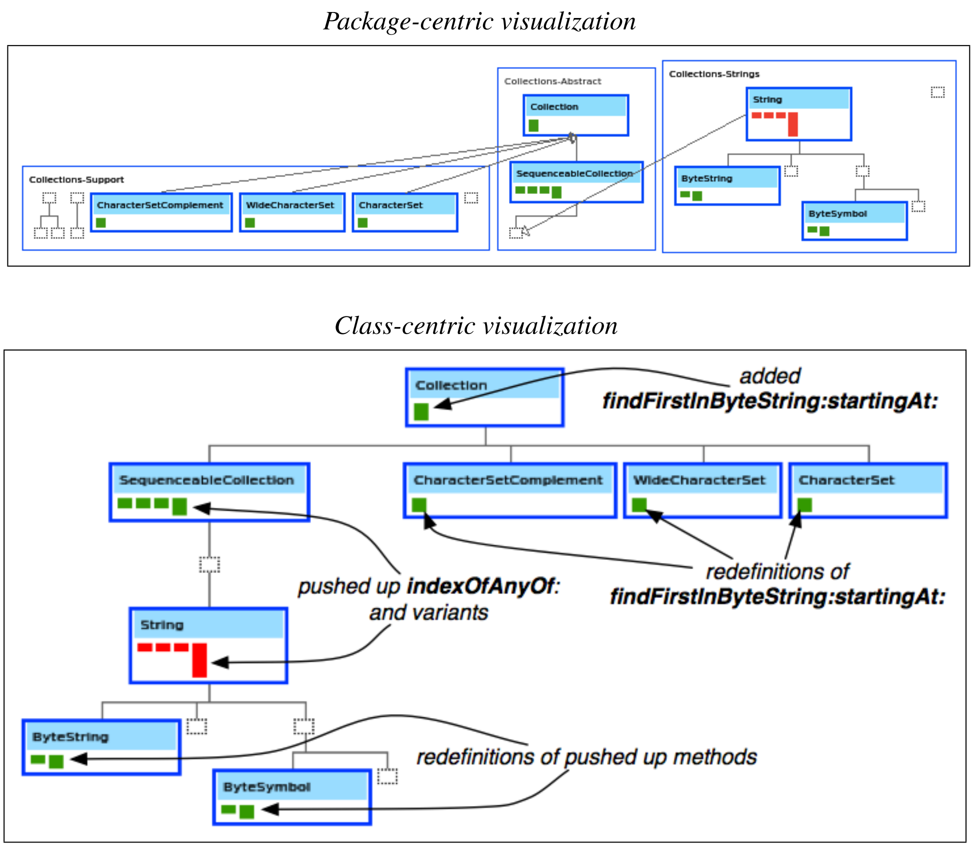

Package-centric visualization
Collections-Abstract
Collections-Strings
Collection
String
SequenceableCollection
Collections-Support
CharacterSetComplement
WideCharacterSet
CharacterSet
ByteString
ByteSymbol
Class-centric visualization
added
findFirstInByteString:startingAt:
pushed up indexOfAnyOf: and variants
redefinitions of findFirstInByteString:startingAt:
redefinitions of pushed up methods

Figure 5.15: *Pushing up methods in the* **SequenceableCollection** *class hierarchy, and introducing a method in the* **Collection** *class hierarchy –* `SLICE-Issue1629-universal-indexOfAnyOf-nice.1` *(Dec. 18th 2009).*

Figure 5.15 shows a push up method refactoring affecting a inheritance tree contained in three packages. At the top a package-centric visualization and at the bottom a class-centric visualization of the same scenario are shown. Note that class-centric visualizations can be more appropriate to observe this kind of changes. The method indexOfAnyOf: and its variants – originally implemented in String (removed methods) – were pushed up to its indirect superclass SequenceableCollection and their redefinitions were added in two subclasses of String (green methods).

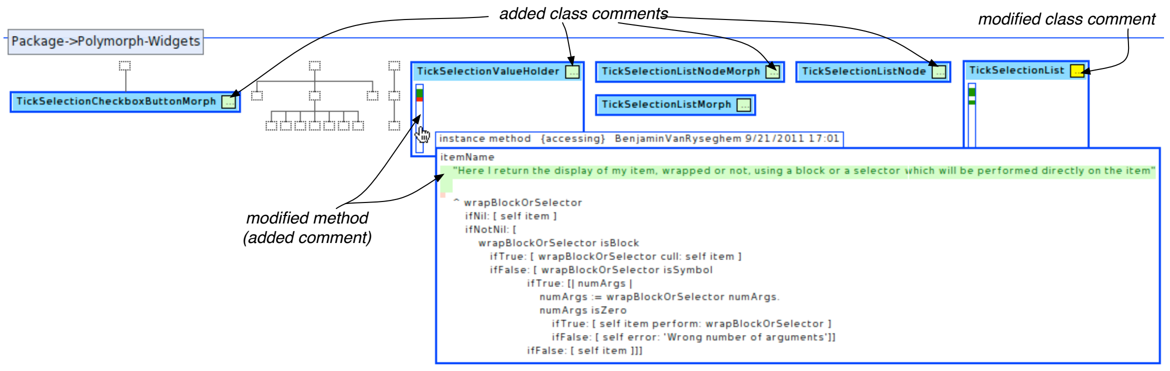
This scenario also shows another change affecting the same inheritance tree: the introduction of findFirstInByteString:startingAt: in Collection, the top superclass of this hierarchy, and its redefinitions (added methods) in CharacterSetComplement, WideCharacterSet and CharacterSet.

The integrator can use any of the visualizations for each scenario. However, he can observe that certain scenarios fit better with a particular kind of visualization. This case for example, as shown in the package-centric visualization displays modified classes (with blue borders) in three packages that are linked by inheritance relationships. Still, this visualization does not give an immediate overview of the inheritance tree. Therefore, a class-centric visualization that omits the package containment relationship (as shown on the bottom) is more appropriate. Note that an integrator can have a first idea of what happened: by using the diff as a fly-by help he can confirm that the affected methods represent a refactoring. Moreover, he can also identify that a second change introducing behavior in the same inheritance tree was submitted in the same commit. Questions such as “When multiple packages are committed at the same time, do I really need to load all of them now, or can I just load/merge with the version of the package I am working on?” can be answered.

In this scenario, the structure of changes plays a key role to ease understanding these changes. By only observing the dashboard an integrator can already infer what happened in the inheritance tree and which other classes within the tree may be affected.

5.6.5 Adding comments

Non-functional changes, such as additions or modifications of class/method comments do not change the semantics of an application. Usually, these changes are distributed over several entities producing large lists of changes. Users of diff tools have to check each change just to find whether it was a cosmetic change, using valuable time for a task that should not demand it. *Torch* presents a class comment as a box next to the class name, and it is displayed in green, red or yellow for an added, removed or modified comment respectively. The users can know that even though a change can be large in terms of the number of modified entities, there is no semantic impact.

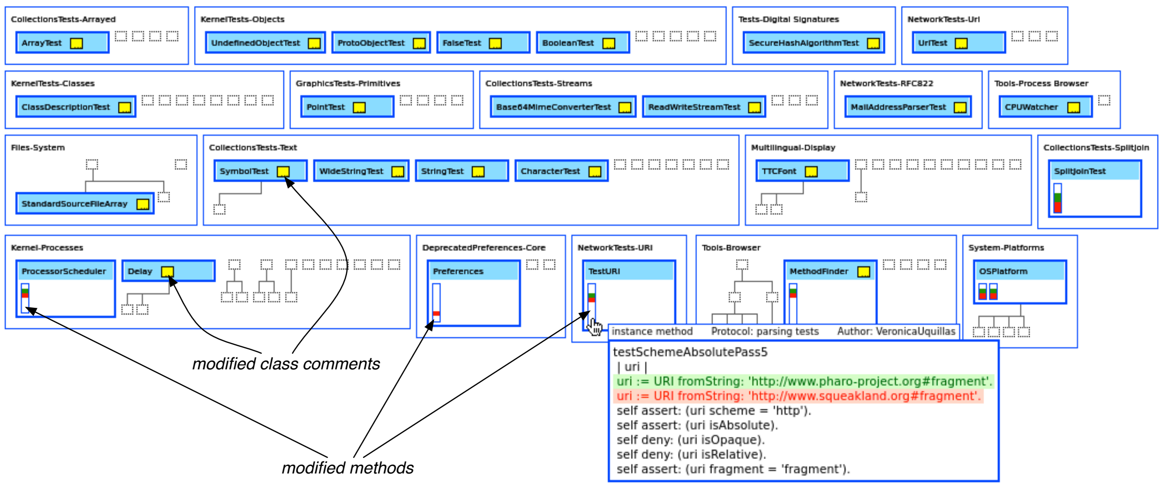


```
instance method   {accessing}   BenjaminVanRyseghem 9/21/2011 17:01
itemName
	"Here I return the display of my item, wrapped or not, using a block or a selector which will be performed directly on the item"

	^ wrapBlockOrSelector
		ifNil: [ self item ]
		ifNotNil: [
			wrapBlockOrSelector isBlock
				ifTrue: [ wrapBlockOrSelector cull: self item ]
				ifFalse: [ wrapBlockOrSelector isSymbol
							ifTrue: [| numArgs |
								numArgs := wrapBlockOrSelector numArgs.
								numArgs isZero
									ifTrue: [ self item perform: wrapBlockOrSelector ]
									ifFalse: [ self error: 'Wrong number of arguments']]
							ifFalse: [ self item ]]]
```

Figure 5.16: *Adding comments: documenting the graphical* ***TickSelection*** *morph classes –* `SLICE-Issue-4844-Add-Comments-On-TickList-Classes-BenjaminVanRyseghem.1` *(Sept. 21st 2011).*

Figure 5.16 shows the addition and modification of comments in the graphical TickSelection classes. Figure 5.17 shows the modification of comments in many classes defined in the Pharo core



instance method Protocol: parsing tests Author: VeronicaUquillas

```
testSchemeAbsolutePass5
 | uri |
 uri := URI fromString: 'http://www.pharo-project.org#fragment'.
 uri := URI fromString: 'http://www.squeakland.org#fragment'.
 self assert: (uri scheme = 'http').
 self assert: (uri isAbsolute).
 self deny: (uri isOpaque).
 self deny: (uri isRelative).
 self assert: (uri fragment = 'fragment').
```

Figure 5.17: *Editing comments: removing the* squeak *word from the Pharo core –* `SLICE-Issue-1795-RemovingSqueakReferences-VeronicaUquillas.1` *(June 11th 2010).*

(64 classes among 41 packages were modified) removing any reference to the word **squeak** (the ancestor of the Pharo system) from the documentation. Both cases also show a couple of comment additions and modifications at method level (two shown with the diff as fly-by help). Basically, in these scenarios the developers documented several classes, which are displayed in the dashboard as the colored boxes next to the class names.

Characterizing changes of class comments by means of colored boxes can speed up understanding of the changes within a delta. More importantly, an integrator can immediately identify that these changes do not affect the semantics of the system and therefore they can be safely integrated. Note that by providing this characterization of changes, an integrator can mainly focus on understanding the changed methods. Again, questions such as “What kind of change is it? (Bugfix/New feature/Refactoring/Documentation)” and in this concrete case, questions about the impact of changes (*e.g.,* “What is the total impact of this change?” or “If I just apply the change, what are the parts of my current system that it will break?”) can be answered.

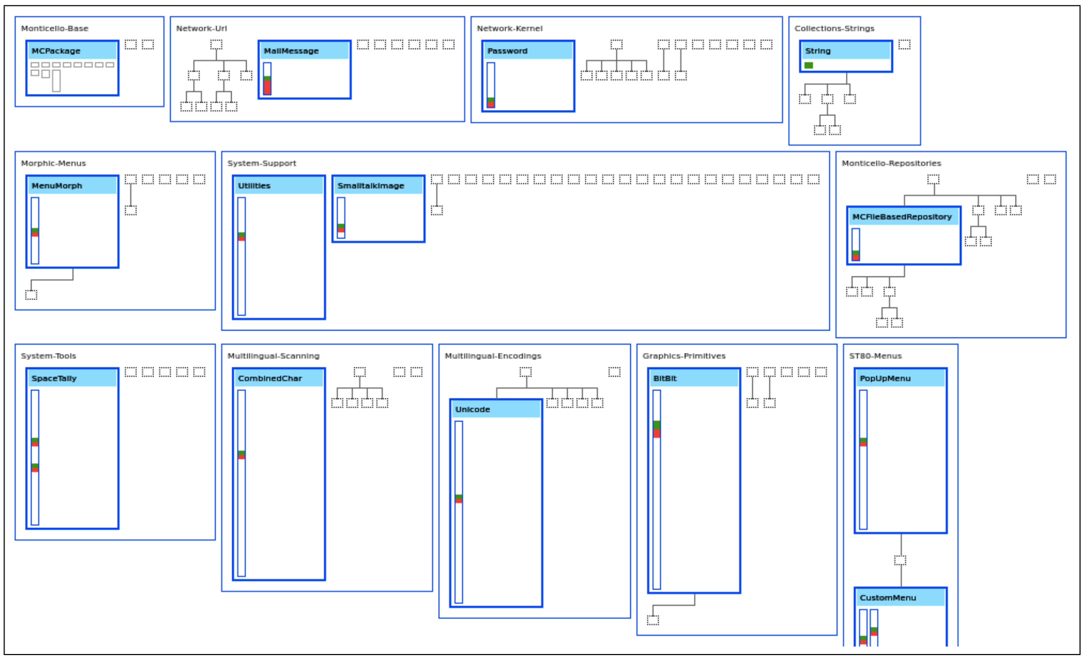
5.6.6 Replacing method calls

Introducing enhancements to a system such as replacing or renaming particular methods, results in a set of changes consisting of both the methods and the clients that call these methods. Depending on how many clients call these methods, a large number of changes may be produced. The integration of this kind of changes can also demand a lot of time from integrators as they will probably inspect every change, even though the change itself is simple.

The package-centric visualization of this scenario will show multiple modifications of methods cross-cutting several packages. As in this case the vocabulary involved in the change (*i.e.,* the old call and new calls) is limited, the use of the symbolic clouds can greatly enhance the visualization of changes. The symbolic clouds aim at showing the relevant vocabulary affected by a change. When replacing or renaming method calls they will show few symbols referring to the old and new calls, each with a high occurrence (*i.e.,* drawn with large font sizes).

Figure 5.20 shows two visualizations applied to the same scenario where a method call was replaced on its clients. At the top a package-centric visualization and at the bottom a symbolic cloud

Package-centric visualization



Mixed symbolic cloud

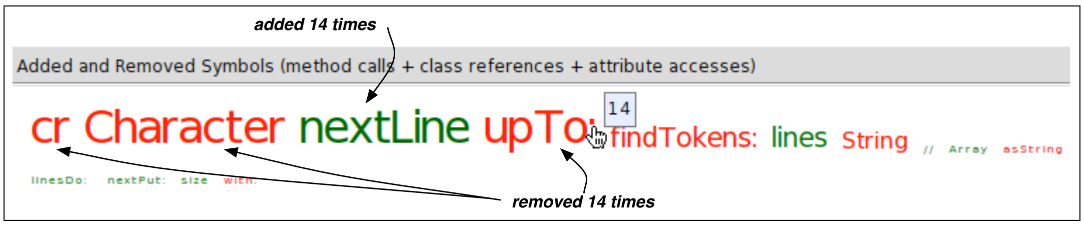


Figure 5.20: *Replacing method calls* **upTo: Character cr** *with* **nextLine** – `SLICE-Issue-2539-useNextLineAndLinesDo-HenrikSperreJohansen.1` *(June 12th 2010).*

are shown. The pattern in the package-centric visualization shows each modified method having two colored sections (corresponding to an added and a removed line of code). The symbolic cloud complements that visual information of program entities by providing the added and removed symbols that actually changed in the code. In the scenario, the mixed symbolic cloud shows that 14 methods were modified. In each method the two combined method calls upTo: and Character cr were replaced by nextLine.

An integrator can observe from the symbolic cloud that the vocabulary involved is small even if the change is large. Moreover, this structural information obtained from the methods' source code is relevant to support answering questions such as "Is the vocabulary used in the change consistent with the one of the system?". By combining the symbolic clouds with the visualization of changes that displays mostly modified methods, the integrator can infer the kind of change.

5.7 Evaluation

In the previous section we provided anecdotal evidence showing that *Torch* supports specific integration tasks. The usage scenarios described in the previous section show us that *Torch* helps integrators and developers understanding and taking decisions when integrating changes. Now the question of knowing whether our approach is useful in practice is an important and difficult one to answer. Indeed it is difficult to perform a *controlled experiment* with master or PhD students since we need experts of complex systems. In addition, accessing a large number of integrators is nearly impossible since integrators are unavailable for performing large experiments. For the first evaluation we performed a limited field study with the integrators of three projects (Moose[6], Pharo, Seaside[7]) from the Smalltalk community. For the second evaluation we performed a pre-experimental user study about the usability of *Torch* with ten developers.

Usability. Nielsen emphasized that usability is not a single, one-dimensional property of a user interface, but that it is associated with five attributes: learnability, efficiency, memorability, errors, and satisfaction [Nielsen 1993]. Another known definition of usability is provided by ISO 9241-11 (Guidance on usability) as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use".

5.7.1 Field Evaluation

Table 5.1 shows the characteristics of the three Smalltalk projects obtained from http://www.squeaksource.com (Projects).

Project	Packages	Classes	Methods	LOC	Versions	Downloads
Moose 4.x	95	599	7186	60359	3434	341031
Pharo 1.x	156	1937	44644	346447	9616	1397493
Seaside 3.0	155	1268	11577	83145	4823	1203350

Table 5.1: *Open-source projects with which Torch was evaluated (on March 13th, 2012).*

Moose is an extensive platform for software and data analysis of several object-oriented programming languages [Nierstrasz 2005]. It offers multiple services ranging from importing and parsing data, to modeling, measuring, querying, mining, and building interactive and visual analysis tools.

Seaside is an open-source framework for developing sophisticated web applications in Smalltalk [Ducasse 2010]. It supports agile development through interactive debugging and unit testing.

We asked two integrators of each of these projects to use *Torch* during their daily work. We also asked them to answer the questionnaire presented in Appendix A. The questionnaire is composed of two main parts. The first part presents closed questions that the integrators needed to mark using a

[6]Moose: http://www.moosetechnology.org
[7]Seaside: http://www.seaside.st

5-point Likert scale: *strongly disagree = 1*, *disagree = 2*, *neither agree nor disagree = 3*, *agree = 4*, and *strongly agree = 5*. The second part consists of open questions oriented to obtain more feedback of what can be improved in *Torch*.

The six integrators agree that change integration is a difficult task (3 rated *agree* and 3 rated *strongly agree*). With respect to their personal qualification, they reported to be expert on the system they integrate, and five of them find visualizations in general very useful. One integrator reported that in general he does not find visualizations useful (he gave a neutral answer on the question *Do you find visualizations useful?*), but after performing the evaluation he reported that the dashboard and its visualizations helped him in the integration process and that he wants to use *Torch* from now on.

In Table 5.2 we present a summary of the results of nine questions of the first part of the questionnaire (see Appendix A). We classify these results in two categories: (a) common tasks that they perform when comparing versions, and (b) their experience using *Torch*. Note that in the second category we only show the questions about the general overview of the use of *Torch*. The values presented in the table correspond to the number of integrators that marked a rating scale.

Question	**Strongly disagree**	**Disagree**	**Neither agree nor disagree**	**Agree**	**Strongly agree**
Tasks when comparing versions					
Identify if changes contain one or multiple fixes	0	1	1	2	2
Identify / characterize changes (semantic, cosmetic, structural fix) or (maintenance, addition, removal, enhancement)	0	0	0	2	4
Assess criticality of changes	0	1	1	1	3
Analyze impact of changes	0	0	0	0	6
Compare branches for merge	0	1	0	1	4
Torch experience					
Would you like to use *Torch* in your daily integration process?	0	0	0	3	3
Does the *Torch* dashboard help you?	0	0	0	3	3
Do you find the diff as a fly-by help showing code on any entity useful?	0	0	0	0	6
Do you think you got a better understanding of the changes, their scope and their impact using *Torch*?	0	0	0	3	3

Table 5.2: *Summary of partial results about the use of Torch.*

The results for the first category show that for integrators the main goal of comparing versions is to assess the impact of changes, followed by the identification and characterization of changes, and the comparison of changes between branches for merging purpose. This confirms again that such activities play a key role when integrating changes.

The results in the second category show that integrators were positive, especially when it comes to using *Torch* in their daily integration process. In particular, they were really positive about the omnipresent diff as a fly-by help. This confirms that integration is a textual activity but that visualizations and textual diffs can be efficiently integrated. Finally, the six integrators also agreed that *Torch* provided them with a better way to understand changes.

The second part of the questionnaire included open questions such as:

- Which features of *Torch* need to be improved?
- Do you think some aspects are not covered by *Torch*? Which features are missing?
- Do you know about existing approaches/tools intended for version comparison presenting visualizations with the structural model and changes as *Torch* does? If yes, mention them.

None of them know about approaches that present an overview of changes using their structural information as *Torch*. This in particular reinforces our knowledge about the lack of support for helping the integration process with other tools than file or folder diffs. Furthermore, they provided us valuable feedback for improvements and missing features. Below we present some of the integrators' recommendations. We consider these as future work. "Torch should...":

- classify changes by semantic impact.
- steer the decision to merge or cherry pick a change directly from *Torch* and not with yet another tool.
- detect simple renamings.
- merge some visualizations and provide the different representation of classes (structural and condensed) on demand.
- include a panel to allow integrators to customize the level of details presented in one visualization, instead of providing multiple visualizations.

Other suggested improvements by the integrators were already implemented in *Torch*, such as:

- show inheritance or trait usage relationships on demand (especially when unchanged classes are linked to changed classes).
- add extra information of a method when showing the diff as a fly-by help of its source code.

5.7.2 Pre-Experimental User Study

We performed a second evaluation to assess the usability of *Torch*. This pre-experimental user study was mostly oriented to Smalltalk developers. Even if this kind of study cannot result in any absolute claim regarding the usability of *Torch*, it provides insights about the perception of users towards the tool and several of its features. Quasi-experiments have been successfully applied for providing an initial assessment of program comprehension tools [Matthijssen 2010].

The definitions of usability before mentioned relate to the perceptions of the participants that we quantify in our evaluation of *Torch* and its features. By means of using *Torch* in concrete scenarios the participants provide their opinions about its usability.

Study design. The quasi-experiment consists of a pre-test and post-test, both presented in Appendixes B and C. The pre-test quantifies the attitude of developers towards tool support and change understanding, and their expectations regarding change visualization tools. The post-test quantifies their perception about the task performed, their experience using *Torch*, and the evaluation of *Torch* and its different features. The results of the post-test give us insights about how the participants feel about the efficiency and satisfaction of using *Torch* to analyze and understand changes.

We compare the results of the pre-test and the post-test to quantify how the use of *Torch* influenced the developers' perception of a visual tool for understanding changes, and which of *Torch*'s features were considered useful by the developers. To this end, we measured the following properties:

- **Value of visualizations:** Do visualizations aid in understanding changes?
- **Information usage:** Does the combination of textual and graphical information speed change exploration and understanding?
- **Class representation:** Do UML like class representations provide a suitable means to express structural characteristics?

Ten developers performed the experiment that took about 40 minutes in total and consisted of the following steps:

1. Fill out the pre-test. This test consisted of 21 statements that, next to measuring the properties aforementioned, asked them about their background knowledge.
2. Attend a short presentation about *Torch* and its features.
3. Use *Torch* to understand two usage scenarios and identify the change patterns in each case.
4. Fill out the post-test consisting of 22 statements.

For the third step, we provided the participants with a Pharo image loaded with *Torch* and the two usage scenarios. We made use of two scenarios presented in Section 5.6. More precisely, the "removing a feature and deprecating its API" and "pushing up methods / introducing methods in a class hierarchy" scenarios, as both show interesting change patterns.

Both tests used a 5-point Likert scale to score each statement: *totally disagree = 1*, *disagree = 2*, *neither agree nor disagree = 3*, *agree = 4*, and *totally agree = 5*.

Developers profile. The ten participants of our user study are experienced software developers with various backgrounds. 2 hold an Engineering degree, 5 hold a Master degree and the remaining 3 a PhD. The initial part of the pre-test asked them about their knowledge of development in general, Pharo usage, IDE usage and version control systems usage.

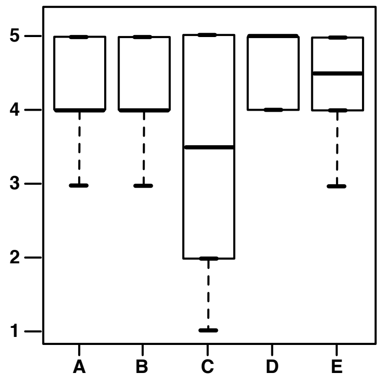


Figure 5.21: *Boxplot of pre-test – participants' background: (A) development experience, (B) OO experience, (C) Smalltalk IDE experience, (D) usage of IDE's facilities, and (E) usage of version control systems' facilities.*

Figure 5.21 shows a summary of the results in this part of the pre-test using boxplots. They give an indication of how the data (*i.e.,* answers) is distributed. As seen in the boxplots for the values of (A) and (B), note that the median in both cases is 4 (*agree* on the scale), the maximum value is 5 (*totally agree*) and at least one developer rated 3 (neutral answer) as the minimal value. Meaning that the majority qualified themselves as experienced OO developers. Regarding the use of a Smalltalk IDE (not necessarily Pharo), the result for (C) shows that our group of participants was not limited to Smalltalk developers. For example, at least one developer has no Smalltalk knowledge, several developers have little knowledge of Smalltalk IDEs and several developers are proficient with Pharo. As it can be seen in (D), all of the developers highly use some facilities of IDEs. Finally, they also regard themselves as knowledgeable users of version control systems. At least one participant rated himself as intermediate user, as shown in (E).

The results in general indicate that our participants were qualified for performing our experiments using *Torch*. The majority are experienced object-oriented developers with a good knowledge of IDEs and version control systems. We consider the fact that not all of them are Smalltalk experts an advantage, because we can receive different points of view regarding the support provided for understanding changes.

Pre-test vs. Post-test. For each of the three properties mentioned before we compare two statements: one from the pre-test and one from the post-test. We show the comparison of the three measures in Figure 5.22. The white bars represent the values from the pre-test and the grey bars represent the values from the post-test.

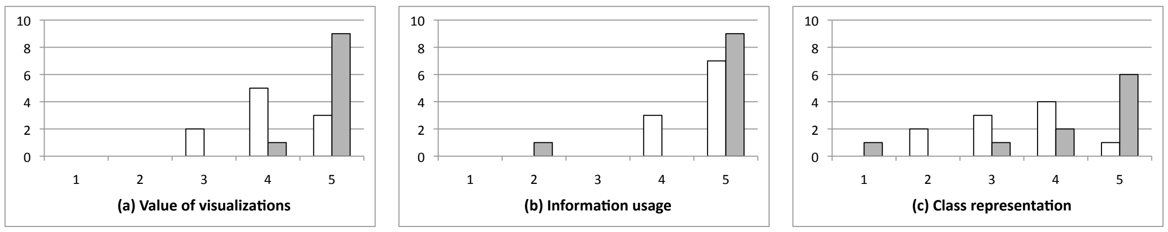


Figure 5.22: *Comparison of the pre-test (shown in white) and post-test (shown in grey). X axis represents the 5-point Likert scale, Y axis represents the number of participants that selected a scale point.*

The *value of visualizations* property shown in Figure 5.22(a) measures the statements "Having a visualization helps understanding changes" and "Torch aids in understanding changes".

The *information usage* property shown in Figure 5.22(b) measures the statements "Graphical and textual information about changes should be combined together to speed change exploration and understanding" and "I like the fly-by help to explore the source code from within the visualizations at any time without opening a new window".

The *class representation* property shown in Figure 5.22(c) measures the statements "I find diagrams of my software (UML, ...) useful" and "Torch's use of class representations (UML like) makes change identification easier".

As we can see in Figure 5.22(a), all developers were very positive about the value provided by *Torch* (see grey bars). One agree and nine of them strongly agree that *Torch* helped them understanding changes. In the pre-test, the results were less positive than in the post-test. This shows that *Torch* exceeded their initial perception of the usefulness of visualization for understanding changes.

For the *information usage* property shown in Figure 5.22(b) all participants (strongly) agreed with the advantages of combining textual and graphical information in the pre-test. Regarding the post-test, only one participant did not agree with the advantage of the diff as a fly-by help as a means to provide textual information on top of the visualizations. Despite being asked, this person did not fill out any details. The rest strongly agree that the fly-by help offers an advantage, confirming that combing both kinds of information enhances the support provided by *Torch*.

With respect to the visual representation of classes shown in Figure 5.22(c), the results were heterogenous in both the pre-test and post-test. The opinion of the developers is divided with respect the usefulness of diagrams such as UML: two disagree, three were neutral, and five (strongly) agree. This perception improved after using *Torch* as shown in the results of the post-test. Only one participant strongly disagreed and one was neutral about the usefulness of a visual representation for classes similar to the one used in UML diagrams. Eight developers (strongly) agreed that this eased change identification.

Features of Torch. From the post-test we extracted relevant information about the usefulness of several features of *Torch*. In particular, we present the developers' perception related to five features: (a) is the detailed class representation useful?, (b) does the package-centric visualizations provide enough information?, (c) is the diff as a fly-by help useful?, (d) is the full class structure as a fly-by help useful?, and (e) is the presence of unchanged program entities needed to understand the context of the changes?. The results are shown as boxplots in Figure 5.23.

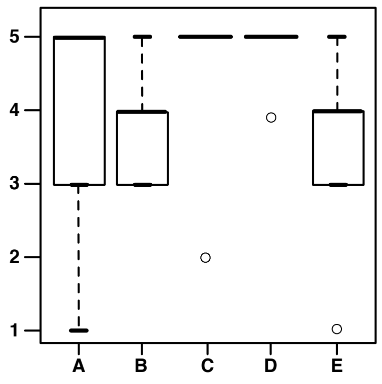


Figure 5.23: *Boxplots of post-test – Torch's features: (A) detailed class representation, (B) package-centric visualizations, (C) diff as a fly-by help, (D) full class structure as a fly-by help, and (E) presence of unchanged entities.*

For the first feature, the majority of the results are positive. The median is 5 corresponding to *totally agree* and the minimum value is 3 corresponding to *neither agree nor disagree*. With respect to package-centric visualizations, the results indicate that developers agreed with the usefulness of this feature and several have a neutral opinion about it. The third boxplot refers to the diff as a fly-by help. It shows that nine developers totally agreed with the advantages of the diff and one disagreed without providing a reason for this. The results for the fourth feature are also very encouraging. All participants saw the advantage of providing this information on top of the visualizations. Finally, the developers were positive about the presence of unchanged entities in the visualizations. As seen on the last boxplot (E) the median is 4 corresponding to *agree*.

5.7.3 Threats to Validity

For the field study, we contacted the integrators by email. To prevent the introduction of any bias regarding the use of *Torch*, we did not interact personally with these integrators. We provided integrators with: (1) a short tutorial about the features of *Torch* and how to use them, (2) the instructions to load *Torch* into a Pharo image, and (3) the questionnaire to be filled out. None of integrators had problems loading or using *Torch*, and all of them were able to try each of its features. This field study provided us with insights about the usability of our tool.

For the pre-experimental user study, we gathered the 10 developers in one room to explain them about the experiment. After they filled out the pre-test, we only provided a short demonstration of *Torch* that took about 10 minutes. For the rest of the experiment, they were left on their own. Each developer went to his respective office to apply *Torch* to two real scenarios and fill out the post-test. As with the field study, we did not want to influence them when using the tool.

Our pre-experimental study does not allow us to make any generalizable claims regarding the usability of *Torch*. Nevertheless, the user study does allow us to observe how potential users perceived our tool. The validity of these observations is however subject to a number of threats.

Performed Tasks. One possible threat to validity of this study is the definition of the set of tasks performed by our participants. Our scenarios showed the benefits of our approach characterizing changes. While there is a chance that these scenarios coincidentally favor *Torch*, we would like to stress that they were not designed for this experiment. Both scenarios came from actual integration activities within the development history of the Pharo project. We described them in Section 5.6 among other usage scenarios. However, we are aware that a more extensive evaluation of our approach is needed and we discuss with future work in Section 8.4.

Size of the group. The number of participants in our pre-experimental user study was small. Ten developers might not form a representative sample to evaluate the usability of the features of *Torch*. However, we believe it to be representative since the background of the participants was diverse. Only 3 participants are researchers, which means the majority evaluated *Torch* with the expectation of an actual tool and not just an experimental approach. They all reported to be experienced developers, and not necessarily Smalltalk developers. Five participants master other languages such as Java or C++. This gave us different points of view regarding the use of *Torch* and its features, considering that they are used to different IDEs such as Eclipse.

Language generalization. The integrators who replied to our first validation are all Smalltalk developers. The developers who participated in the second validation formed a rather heterogeneous group. However, most of them use Smalltalk for their work. We did not test *Torch* on Java or C# programs and with integrators. Since the file structure of these languages is also based on packages, classes, methods, and since the *RingS* meta-model used by *Torch* can be extended we believe that our approach can be adapted to other languages.

Generalization. As with any field study, it is difficult to conclude that our approach can be fully generalized. The projects we selected are real open-source projects with a large number of versions. They are heavily maintained and developed. We would like to investigate if *Torch* can be efficiently used for Java applications, however we are concerned with the engineering cost of integrating *Torch* in the Eclipse/Idea IDEs in addition to the Smalltalk ones.

5.7.4 Discussion

The evaluations performed with integrators and developers were very productive. They not only gave us feedback regarding the usability of our approach but also provided us with valuable ideas for improvements from the point of view of (potential) users.

In the case of the field study, the six integrators filled out every single open question with various suggestions of what can be improved, added or even removed. In general, they all found *Torch* useful for their respective integration tasks. Even though, not all of them are used to deal with visualizations as a means to support tasks, after using *Torch* they agreed that the dashboard eased the understanding of changes. We were very satisfied after knowing that the integrator reluctant about the value of visualizations requested to have an image with the tool to use it for his work.

In the case of the pre-experimental user study, we could evaluate the perception of potential users of *Torch*. Having an heterogenous group, with no just Smalltalk developers was an advantage. As we could gather the perception of developers regarding the usability of a tool that supports change understanding from different points of view. This study also provided us with in-depth insights about every single feature in *Torch* and what can be improved.

Furthermore, while applying *Torch* to our usage scenarios described in Section 5.6 we can see patterns in the visualizations that speed up understanding of changes, for example, feature removals, cross-cutting changes, documentation changes, and so on. This can guide integrators during the comprehension of changes, *e.g.,* they can focus on analyzing particular changes that do not fit within the visual pattern. The usage scenarios also served us to detect possible cases that decrease the level of help provided to integrators and developers. In the following, we present three aspects that should be considered for improving our approach.

- When commits are messy and contain unrelated code, *Torch* presents the situation as it is. Currently, it does not support tagging to classify changes. Being able to tag changes into a kind of slices (*i.e.,* separate groups of changes) would help in this situation.

- In the same vein, due to the fact that *Torch* allows the simultaneous comparison of multiple pairs of versions (*e.g.,* all the package versions involved in a commit), this may result in a complex visualization with a high number of drawn entities and inheritance relationships. For example, if changed classes have a considerable number of subclasses cross-cutting several packages, the edges representing inter-package inheritance relationships produce noise. One integrator pointed out this problem and gave some ideas of improvements, which we have since taken into account.

- The most important limitation of *Torch* is that it only shows structural information. How an integrator or developer understands the impact in terms of *different program behavior* is also very important. We are aware that assessing the impact of a change on the program behavior is needed but at the same time it is a difficult task since it is another step towards semantic merge or understanding program semantics. We address this point as an avenue for future work in Section 8.4.

5.8 Related Work

In Section 3.5 we presented the state-of-the-art including several approaches that support understanding changes. After explaining *Torch* we can provide a more in-depth look of how these approaches differ from *Torch*. In the following, we start by presenting a summary of approaches in the area of software visualization, then we proceed to a more extended discussion related to approaches that characterize, understand and document changes, and provide aspect analyses.

5.8.1 Software Visualization

Within the reverse engineering and software maintenance community, software visualizations are a well-established medium for supporting tasks related to program comprehension and evolution. Tasks that benefit from software visualization include software exploration [Storey 1997b, Storey 1997a], visualization of metrics [Lanza 2004, Arbuckle 2008], visualization of co-changes [Beyer 2005], the study of evolution patterns [Lanza 2001, D'Ambros 2006, D'Ambros 2007], and the comprehension of individual classes [Ducasse 2005, Robbes 2005] and packages [Ducasse 2006a]. Beyond source code, visualizations have been proposed for other types of data such as bug tracking information [D'Ambros 2007] and versioning information [Lungu 2010] as well as aspect-oriented programming [Fabry 2011].

However, none of these approaches target aiding developers in comprehending how particular changes affect a software system, as is needed for cherry picking changes and therefore in assisting integration.

5.8.2 Class and Method Understanding

Ducasse and Lanza [Ducasse 2005] provide a call-flow based representation of classes to support class understanding. Their approach – Class Blueprints – is a semantically augmented visualization that shows the internal structure of a class distributed by showing different layers that group methods and attributes. Another visualization approach – Microprints – is proposed by Robbes *et al.* [Robbes 2005]. It offers three pixel-based visual representations of methods enriched with semantic information such as state access, control flow, and invocation relationship. This approach provides fine-grained information about the method signature and body for supporting method understanding.

While both approaches provide deep understanding of program entities (classes and methods), they do not provide the same information as *Torch* necessary for characterizing changes within a delta. *Torch* eases in understanding changes in a single delta regarding to the structure of the system. *Torch* could be enhanced by integrating Class blueprints and Microprints into the changed classes and methods.

5.8.3 Change Characterization

Dragan *et al.* [Dragan 2011] propose a technique to characterize a commit based on the methods that were added or removed in that commit. This approach is influenced by their previous works on revealing patterns of design from the current version of the system at three different levels of abstraction: method [Dragan 2006], class [Dragan 2010], and system [Dragan 2009]. Such a categorization of methods (stereotypes) [Dragan 2006] takes various properties of the method (accessing data, changing state, interaction with other objects, and so on) into account. By detecting these method stereotypes and, by studying the distribution of the method stereotypes within a commit, they propose a number of categories of different kinds of commits. This approach is related to our work in the sense that the identified commit types can provide an integrator with valuable information regarding the size and scope of a commit. However, this technique only takes into account the changes that might impact the system's design and therefore does not provide a general overview of the changes, or a complete categorization of changes prior the integration phase.

5.8.4 Understanding Changes

Fritz and Murphy [Fritz 2010] present a study in which they interviewed developers regarding the different kinds of questions they need answered during development. Alongside this study, they introduce the information fragment model and associated prototype tool for answering the identified questions. This model provides a representation that correlates various software artifacts (source code, work items, teams, comments, and so on). By browsing the model, developers can find answers to particular development questions.

While a number of the questions that developers need answered during development align with those they need answered during integration of changes, the information fragment model is purely textual and does not provide visualizations of the changes related to the structure of the system.

Several change impact analysis approaches presented in Section 3.5, such as Chianti [Ren 2004], decompose the difference between two versions of a software project into a set of atomic changes. Most of them report change impact in terms of affected (regression or unit) tests whose behavior may have been modified by the applied changes.

While both approaches provide a means to better understand changes, *Torch* offers visual overviews and characterization of changes. This could be complemented with a change impact analysis similar to the one provided by Chianti.

5.8.5 Documenting Changes

Commit 2.0 [D'Ambros 2010] is a tool that supports documentation of software changes at commit time. Using visualizations, the tool allows developers to enrich commit comments with annotations. While similar to our approach, their visualizations are less detailed and contain less information about the changes.

5.8.6 Aspect Analysis

Pfeiffer and Gurd [Pfeiffer 2006] propose Asbro, a tool that provides a tree map visualization of where aspects apply in packages and types. Rectangles representing classes or packages are colored with an aspect color if an aspect applies there. The authors assess their tool as being beneficial for obtaining a high-level overview of aspect application, and state that it is scalable up to on average 2100 classes. Coelho and Murphy propose ActiveAspect [Coelho 2006], a tool that shows an automatically selected subset of the elements in the code, depending on the current focus of the developer. They extend UML with a representation of aspects, method execution advice and method call advice.

Fabry *et al.* [Fabry 2011] propose a visualization tool – AspectMaps – that shows implicit invocations in the source code by visualizing join point shadows where aspects are specified to execute. It provides fine-grained information (*e.g.,* type of advice, specified precedences) for any joint point shadow. AspectMaps allows users to obtain more information of the structure of the code by using a selective structural zooming functionality.

Asbro, ActiveAspect and AspectMaps are dedicated to the visualization of aspects and do not support diff of source code or removal/changed code support or author information.

5.9 Conclusion

In this chapter we presented *Torch*, our tool for providing a characterization of changes within a single delta and supporting change understanding. *Torch* offers a change overview by means of visualizations, change metrics and two omnipresent contextual fly-by helps. Furthermore, *Torch* offers a diff to explore the source code of a method, and a visual representation to explore the complete structure of a class in the dashboard.

First, we explained the layout and main components of *Torch*, along with the different kinds of visualizations available: package-centric, class-centric and symbolic clouds.

Second, we presented the *RingS* single-delta change meta-model that allows us to provide descriptive and structural information of program entities within a target version and of the changes regarding to a base version.

Third, we described our evaluation of *Torch* by means of six usage scenarios extracted from the development history of Pharo. By means of the usage scenarios, we have presented the capabilities of *Torch* for charactering changes, defining change context and overview. Moreover, these scenarios allows us to visualize change patterns on the visualizations, such as feature removal or method replacement.

Fourth, we described two studies as part of the evaluation of *Torch*. (a) A limited field study performed with six integrators of 3 projects within the Smalltalk community. Integrators applied *Torch* to their daily integration activities and they filled out a questionnaire. This study provided us with valuable feedback regarding the usability of *Torch* and with suggestions for improvement. (b) A pre-experimental user study by means of pre-tests and post-tests performed by 10 developers. They applied *Torch* to two of our usage scenarios. We did not make generalizable claims out of the results, but they have provided us with insights into how users perceive *Torch* and several of its features.

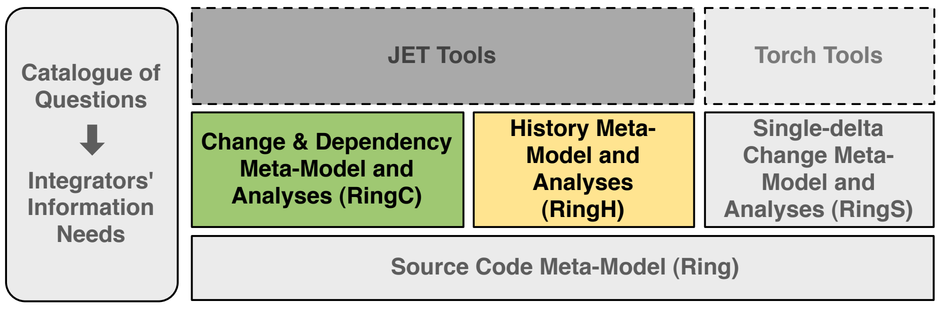
Fifth, we discussed several approaches related to *Torch* in the areas of software visualization, class and method understanding, change characterization, change understanding and aspect analysis.

CHAPTER 6

RingH and RingC: History and Change Models & Analyses

Contents

Contributions Map


Catalogue of Questions
Integrators' Information Needs
JET Tools
Torch Tools
Change & Dependency Meta-Model and Analyses (RingC)
History Meta-Model and Analyses (RingH)
Single-delta Change Meta-Model and Analyses (RingS)
Source Code Meta-Model (Ring)

Overview

This chapter presents two meta-models – *RingH* and *RingC*– that support the analysis of systems' histories and serve as the underlying models for the version comparisons and streams of changes analyses. Both models are built on top of the *Ring* source code meta-model described in Chapter 4. First, we describe existing meta-models that model the evolution of Smalltalk systems. Second, we present the history meta-model *RingH* and how the history of a system stored in a versioning repository is built in terms of *RingH*. Third, we present the change and dependency meta-model *RingC* and how a change model is created. Fourth, we explain the delta and dependency mechanisms that compute deltas and dependencies. In this chapter we present *RingC* and the delta and dependency analyses at a contextual level. In the next chapter, we present a client tool built on top of *RingC* and also provide some benchmarks regarding the efficiency and size of the model and analyses.

6.1 Modeling the Evolution of a System

In Section 3.5, we briefly presented several approaches modeling source, changes and history. Later in Chapter 4, we discussed in-depth several meta-models for source code representation, and we introduced our own unified source code meta-model, namely *Ring*. This meta-model is the basis of our infrastructure and serves as foundation for modeling the history and changes of a system.

In this section, we revisit two Smalltalk approaches for history and multiple version representation introduced in Section 3.5. Both approaches extend the FAMIX source code meta-model that we discussed in Section 4.4.1.2.

6.1.1 Hismo

Hismo [Gîrba 2005a, Gîrba 2006] is a meta-model based on the transformation of a structural meta-model into a history-aware meta-model for software evolution analysis. Key to this meta-model is that the history needs to be modeled as a first-class entity for being accessed and manipulated by other tools. Various analyses have been implemented based on Hismo: for example in [Gîrba 2005b] it was used to assess the evolution of class hierarchies.

Figure 6.1 and Figure 6.2 illustrate part of the Hismo meta-model.

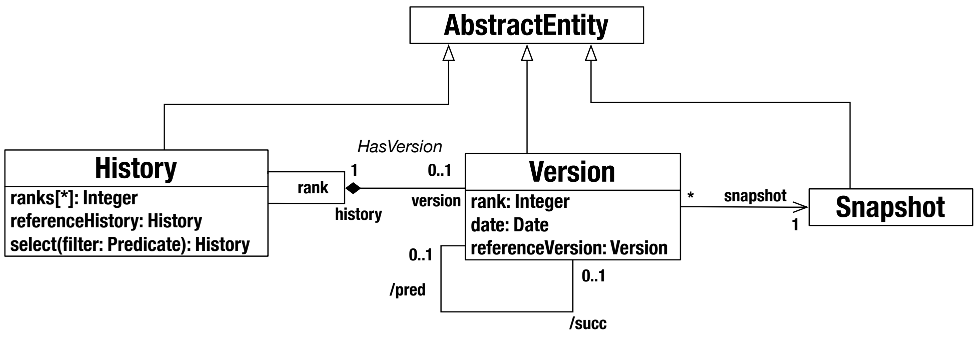


Figure 6.1: *Hismo design*

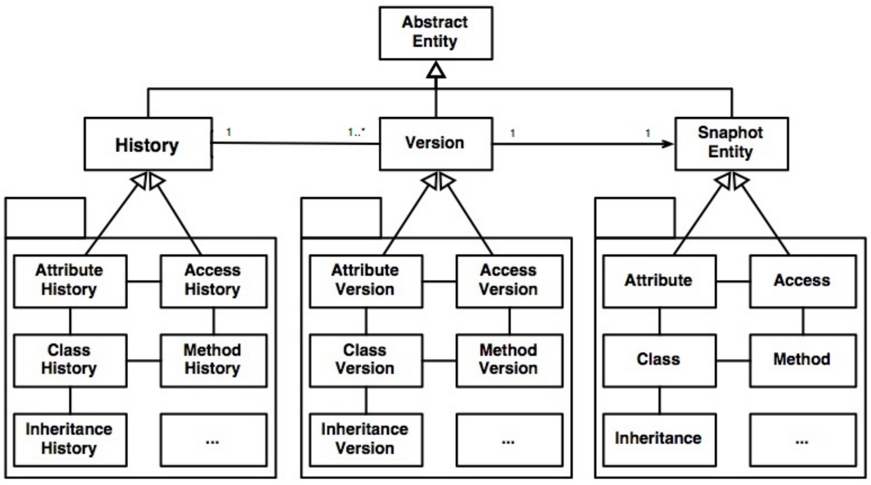


Figure 6.2: *Overview of the Hismo meta-model*

Hismo is based on the notion of history as a sequence of versions. Given a representation of a Snapshot (*i.e.,* an entity at a point int time), time information is added to it through a Version. Moreover, a Version exists in the context of a History. Hismo models are constructed by transforming snapshot meta-models, *e.g.,* FAMIX core meta-models, where each FAMIX model represents a single version of the system.

As shown in Figure 6.2, Hismo uses a class hierarchy to model the different entities and relations in the version history. Considering that the Snapshots are modeled using the FAMIX core, Hismo offers a means to represent the versions of entities that can be found in such a meta-model. More precisely, it models the history of packages, namespaces, classes, methods and attributes, along with the inheritance associations that are defined between the classes.

In a Hismo model, the history of each entity (e.g. class, method, inheritance relationship) is represented by a single history object, that contains a representation of all the versions of this entity. For example, in the figure we can see ClassHistory that is a representation of the history of one particular Class. Within such a class history, each version of the class is represented by a Version object (for the version of a class, this object is the ClassVersion).

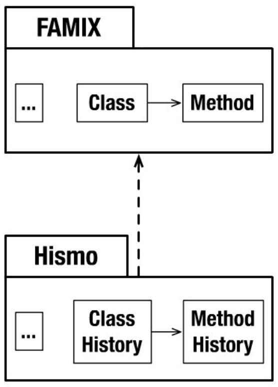


Figure 6.3: *Transforming a FAMIX core meta-model (snapshot) into a Hismo model*

Note that the Hismo model is dual to the FAMIX core meta-model, as shown in Figure 6.3. In other words, if there exists a relationship between two entities (e.g. an inheritance relationship between two classes), this also implies that there will be a relationship between the versions of these classes in the Hismo model.

Hismo is limited because it is a copy-based approach. It does not matter that a program entity only changed a few times in the lifetime of a system: if the system's history is made of 100 versions, Hismo creates 100 Version objects for that particular entity. Therefore, the memory consumption and object creation time is related to the size of the system and the size of its history.

Now consider a concrete case – the history of the Monticello[1] distribution in Pharo 1.3. The core package of Monticello is defined by 98 classes and 1080 methods. The history of this package is made of 143 versions. To ease the calculation of potential created objects by Hismo assume that these classes and methods existed since the initial version. The resulting Hismo model will contain *(143 * 1080) 154440* MethodVersion objects, *(143 * 98) 14014* ClassVersion objects, and the same applies for attributes, accesses, inheritance, and so on.

[1]Monticello in the released Pharo 1.3 (located at http://www.squeaksource.com/Pharo) data on June 25th, 2012

This is a small case compared to the history of Pharo until version 1.3. The history of the Pharo's core is made up of 9622 versions, and the core itself is defined by 156 packages, 2020 classes and 45779 methods. Hismo models for large systems do not scale and are impractical for performing history analysis.

6.1.2 Orion

Orion [Laval 2009, Laval 2011] is an interactive prototyping tool for *software reengineering* that allows developers to simulate changes and compare their impact on multiple versions of software source code models. Orion's meta-model is an extension of the FAMIX source code meta-model. The key point of this meta-model is the memory usage optimization of multiple versions for large models, that is, to save memory space and creation time, entities which do not change are shared between different versions of the model.

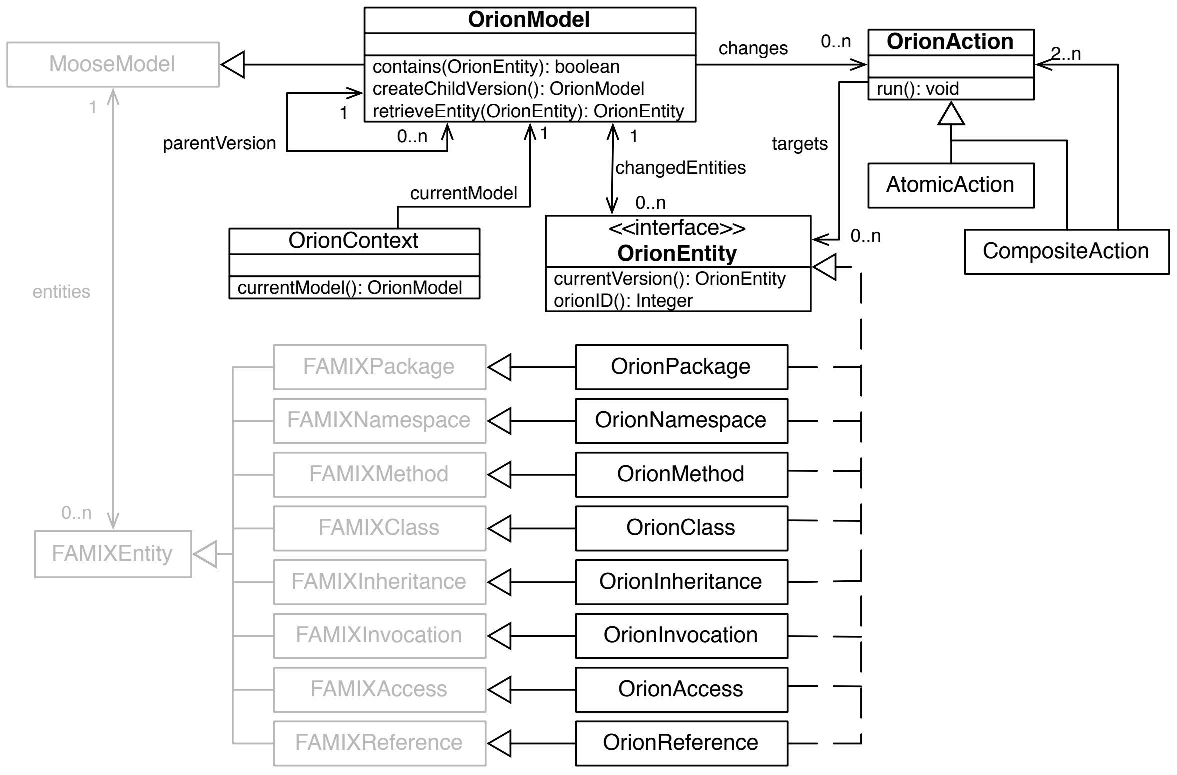


Figure 6.4: *Orion meta-model*

Figure 6.4 shows an overview of the Orion meta-model. OrionModel, OrionEntity, and OrionAction are the core classes in this model.

OrionModel models a version of the system. Each version knows its parentVersion which allows to build a tree structure of the system's history. The tree root represents the original model and contains the program entities from the current source code. An OrionModel keeps a set of OrionEntity objects. OrionContext points to the current version of the system where the reengineering is happening. Thus, navigating between versions implies changing the OrionContext to point to the wanted version.

OrionEntity represents a structural program entity or a reified association between entities. Four kinds of entities are supported: OrionClass, OrionMethod, OrionPackage and OrionNamespace, and four kinds of associations: OrionReference (from a method to a class), OrionInvocation (method call), OrionInheritance, and OrionAccess (from a method to a attribute). Each OrionEntity has an orionID which is unique across all versions. A newly created entity receives a new, unique orionID. A changed entity keeps the same orionID as its ancestor. This identifier allows Orion to keep track of changed entities between different versions of the system.

OrionAction models two kinds of actions – AtomicAction and CompositeAction – that can be performed during the reengineering. AtomicAction such as *remove a method*, *move a class*, or *create a package*, and CompositeAction such as *merge two packages* or *split a class* are supported.

The sharing of entities between models happens when importing the history of a system. That is, OrionEntity objects are only created when a program entity or association actually changed. This combined with the usage of an unique identifier for each program entity or association throughout the models representing a system's history optimizes the memory usage and querying time.

While Orion tackles the disadvantages of Hismo by introducing the notion of shared entities between versions, Orion has been designed to support reengineering. Concretely, it simulates scenarios by means of actions to create snapshots that represent the *future* of a system. Moreover, Orion does not handle explicit changes and dependencies. Both issues limit the applicability of Orion in our context. We focus on representing the *history*, *changes* and *dependencies* to support streams of changes analyses and thus assist the integration process.

6.2 RingH: a History Meta-Model and Analyses

In this section we present our history meta-model and analyses, *RingH*, that supports the representation and analyses of system histories. We mention the requirements for our meta-model before describing its architecture and analyses.

6.2.1 Requirements for RingH

We established several requirements for representing the history that enable us to provide a complete and efficient support for performing analyses.

- *Model history as a first-class entity*: reify the history to encapsulate knowledge about evolution and version information. This allows using historical information to understand the evolution of a system.

- *Allow fine-grained representation*: more than representing packages, classes or methods, we also require to track the history of attributes, class inheritances, and relationships existing in a method's body such as method calls, reference to classes, and accesses to attributes.

- *Use a unique identifier for program entities and relationships*: the use of a unique identifier for the same program entity throughout the history allows to keep track of the whole evolution of this entity and ease the access to such information.

- *Model only changed program entities and relationships*: the history of an entity should only be represented by objects defining a different state of its evolution. Disregarding the number of versions of which the history consists. For example, if a method only changed once in a system made of 15 versions, the method history should be represented by two method definitions (*i.e.,* one for the original method, one for the modified method) shared among the rest of the versions.

- *Keep track of deleted entities*: identify that an entity introduced in V_n already existed in a previous version makes the history more complete. It could be important for an integrator to identify why an entity was removed and later reintroduced.
- *Represent multiple branches and merges*: with a heavy use of branching, the history of a system may be scattered between branches. We need to take this into account and be able to model and track merge operations.
- *Ease of querying*: for large and complex histories an important requirement is to ease the access to such information for integrators.

Furthermore, for representing large systems with large histories two technical requirements should be taken into account: (a) minimal object creation time, and (b) optimized memory consumption. Note that both technical requirements are the motivation for the meta-model of the Orion tool presented in Section 6.1.2. They can be tackled by the requirements *use a unique identifier for program entities and relationships* and *model only changed program entities and relationships*, following the same approach taken by Orion.

6.2.2 Architecture of RingH

Based on the requirements we define *RingH*, a meta-model to provide a historical representation of the program entities and their relationships conforming a system's history. *RingH* is inspired by Orion's meta-model and built on top of the *Ring* source code meta-model presented in Chapter 4.

RingH models the history program entities such as *packages*, *classes*, *traits*, *methods* and *attributes*, and the history of relationships such as *attribute accesses* (method-to-attribute), *method calls* (method-to-method), *class references* (method-to-class) and *class inheritances* (class-to-class or class-to-classes).

The classes in *RingH* are extensions of the classes in *Ring*. This allows us to take advantage of the API and behavior supported by the program definitions and relationships defined in the source code meta-model.

Figure 6.5 shows the main classes of the history meta-model (shown without background) and how these classes are linked to the main classes of the source code meta-model (shown with grey background).

Core Classes of RingH

In the following, we provide a brief description of the main classes defined in the meta-model. We group them based by their purpose.

Base Class. As Smalltalk does not support multiple inheritance and the classes of the history meta-model already inherit from the correspondent classes of the source code meta-model, we needed another mechanism to support the historical behavior of each class without duplicating code. We implemented several traits, RGTHistory being the core trait in the history meta-model.

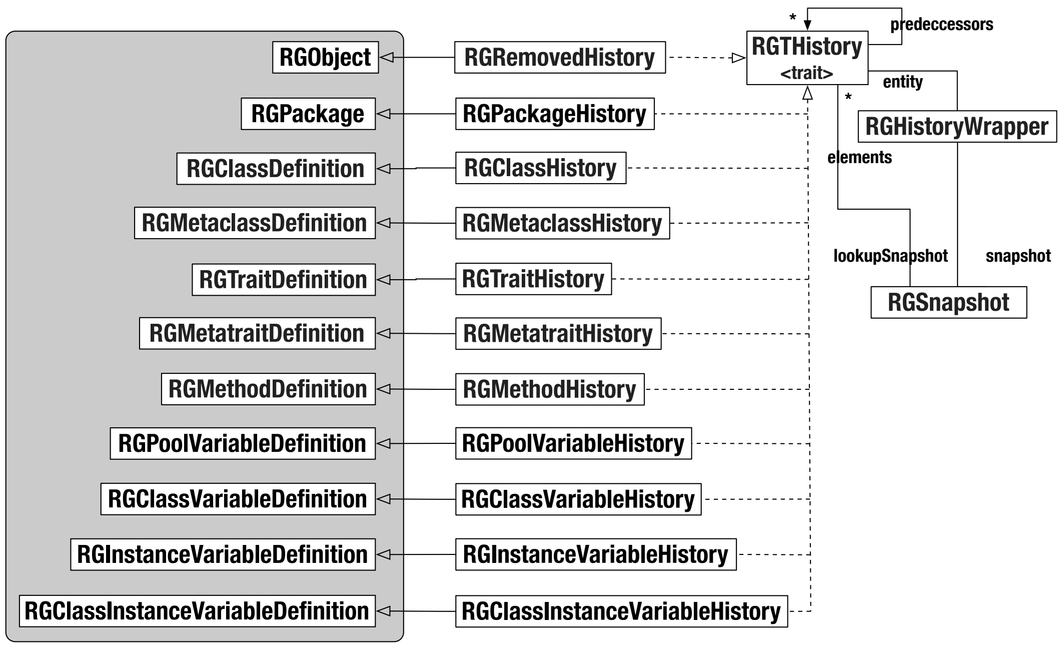


Figure 6.5: *RingH history meta-model - Key classes modeling program entities.*

A trait is a collection of methods that can be included in the behavior of a class without the need for inheritance. This makes it easy for classes to have a unique superclass, yet still share useful methods with otherwise unrelated classes.

RGTHistory defines the behavior that applies to every historical entity in the system. Note in Figure 6.5 that all the classes connect to RGTHistory by means of a dashed edged line which indicates that a class *uses* the behavior defined in a trait. RGTHistory allows classes in the history to define and access key information, such as the unique identifier (id) described in the requirements. The predecessors and successors of an entity that show how that entity evolved (*e.g.,* if the history of class Zoo is represented by Zoo → Zoo' → Zoo" where each of them have the same id, then the predecessor of Zoo' is Zoo and its successor is Zoo". The snapshot in which each object is created (*e.g.,* the class Zoo was created in the first snapshot S_1). The snapshot in which each object may be resolved (*e.g.,* if the class Zoo" was created in the third snapshot S_3 and it is be resolved in S_1 the result is the Zoo object). Additionally, RGTHistory provides key utilities for comparison, evolution support, querying, testing and so on.

At the end of this section we describe snapshots in detail. Meanwhile, a snapshot can be conceptually considered to be the version of the system in which an entity changed with respect to its previous state.

Special Classes. In the model we define three special classes: RGNilHistory, RGRemovedHistory and RGHistoryWrapper. Here we describe the two first classes that play an important role in the evolution of any program entity. RGHistoryWrapper is explained together with snapshots at the end of this section.

RGNilHistory is a singleton used to represent the beginning of the evolution of any program entity or relationship. It can be considered like the nil object in Smalltalk. For example, the history of class Zoo is then represented by:

instance of RGNilHistory $\longrightarrow$ Zoo $\longrightarrow$ Zoo' $\longrightarrow$ Zoo"

Where the predecessor of Zoo is a RGNilHistory object. The class RGNilHistory has minimal behavior that allows us to keep consistency regarding to the used API, and at the same time to keep a consistent representation of the objects conforming to the evolution of any program entity or relationship.

RGRemovedHistory represents a removed program entity or relationship in the history of a system. By means of this class we tackle the requirement *keep track of deleted entities*. This also helps detect cases when a deleted entity is later reintroduced, and at the same time allows us to keep track of the *whole* evolution of that entity. For example, if class Zoo" is removed in the snapshot S_5, the new history of this class is represented by

instance of RGNilHistory $\longrightarrow$ Zoo $\longrightarrow$ Zoo' $\longrightarrow$ Zoo" $\longrightarrow$ RGRemovedHistory $_{Zoo''}$

Now suppose that in a later snapshot the class Zoo was added again to the system, the new representation Zoo"' is linked with its previous existing history resulting in

instance of RGNilHistory $\longrightarrow$ Zoo $\longrightarrow$ Zoo' $\longrightarrow$ Zoo" $\longrightarrow$ RGRemovedHistory $_{Zoo''}$ $\longrightarrow$ Zoo"'

At the implementation level, RGRemovedHistory wraps the last existing entity (Zoo") and the last snapshot(s) in which that entity existed. Note that Zoo" was created in S_3 but existed till S_4. A RGRemovedHistory object is the successor of the last existing program entity, and if that program entity is later reintroduced then the RGRemovedHistory object is its corresponding predecessor. *RingH* treats a removed object somehow as any other historical object.

Representation of Program Entities. *RingH* models the evolution of packages, classes, traits, methods and attributes (known as variables in Smalltalk) as shown in Figure 6.5. Modeling these program entities is part of the information needed to characterize changes discussed in Section 3.4 regarding the structural information and in particular *kind of entities*. Therefore, their history can be used to answer several of the questions from the catalogue presented in Section 3.3.2. In the following, we describe the different classes of *RingH* that model the evolution of program entities.

Classes and Traits. The concrete definitions that model the evolution of classes and traits are RGClassHistory and RGTraitHistory. They inherit from RGClassDefinition and RGTraitDefinition, both defined in the *Ring* source code meta-model. The historical behavior of classes and traits is defined by means of traits. RGTBehaviorHistory is the trait that defines their common behavior and allows them to manage the history of methods. Another trait RGTClassDescriptionHistory used by RGClassDefinition provides the behavior to manage the history of attributes (*i.e.,* variables). Furthermore, *RingH* also models the evolution of metaclasses (inherent to Smalltalk) as first-class entities. Instead of mixing the methods and variables defined at the class-side of a class, and the methods defined at the class-side of a trait, we model metaclasses as separate entities by means of RGMetaclassHistory and RGMetatraitHistory respectively. Therefore, the evolution of a class and its metaclass is maintained separately, but connected at the same time by specifying which is the metaclass of a class, and which is the class of a metaclass. Both definitions always exist together upon evolution.

Methods. The evolution of methods is represented by the RGMethodHistory class that inherits from RGMethodDefinition. A method knows in which class, trait or metaclass it is defined, and in which package it is contained.

Variables. Regarding variables, in Smalltalk there exist 3 kinds of variables that can be defined in a class (*i.e.,* instance variables, class variables, and pool variables also known as pool dictionaries), and one kind of variable that can be defined in a metaclass (*i.e.,* class instance variables). *RingH* models the evolution of these kinds of variables with RGInstanceVariableHistory, RGClassVariableHistory, RGPoolVariableHistory and RGClassInstanceVariableHistory respectively. These four classes inherit from their corresponding definitions in the *Ring* source code meta-model (see Figure 6.5).

Packages. *RingH* also models the evolution of packages with the class RGPackageHistory. A package knows which classes and methods it contains, and classes and methods know in which packages they are contained. Moreover, a RGPackageHistory object also contains versioning information corresponding to its committed package version, such as commit message, committer, timestamp, version name, location (repository), etc. Even though a package is not really considered a source code program entity but instead it is used for deployment, our meta-model treats packages as another program entity to ease their representation and querying. This also reflects the fact that integrators require information of packages as they do with any other program entity. Hence, keeping the history of packages as first-class entities can be used to answer questions and to characterize changes as well.

Representation of Relationships. *RingH* also models the evolution of five kinds of relationships: class inheritances (class-inherits-class) and (class-is-inherited-by-subclasses), method calls (method-calls-method) (known as invocations in Smalltalk), class references (method-refers-class), and variable accesses (method-accesses-variable). We refer to the first two relationships as *class' relationships*, and to the last three kinds of relationships as *method's relationships*.

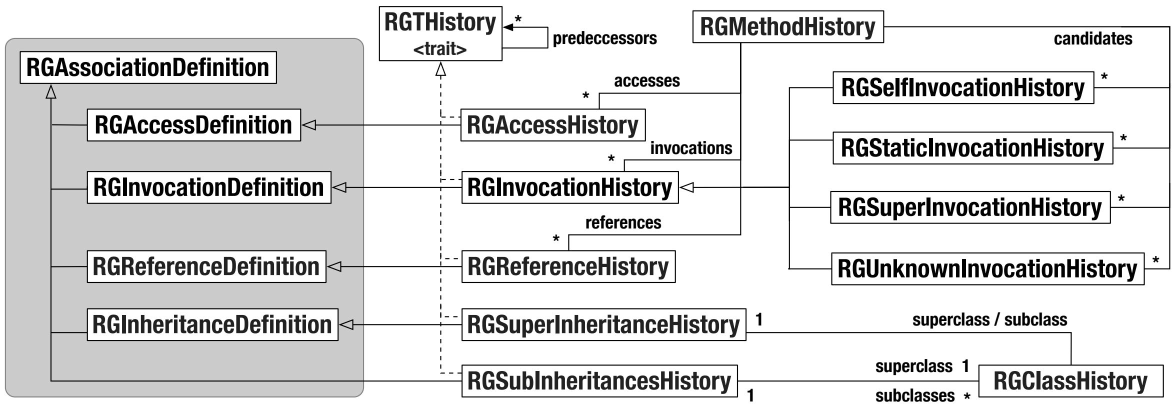


Figure 6.6: *RingH history meta-model - Classes modeling associations*

Figure 6.6 shows the different relationships (shown without background) and how they extend from classes defined in the *Ring* source code meta-model (shown with grey background).

Modeling these relationships allows us to have the history of a system at a fine-grained level, which is one of the requirements established for *RingH*. Class' relationships are part of the structural information needed to characterize changes, and method's relationships serve to identify the structural dependencies at method level (*e.g.,* method foo refers to class Zoo therefore it depends on class Zoo). Relationships are fundamental to analyze streams of changes because they can be used to establish the dependencies of a change within a stream, which in turn enables the characterization of changes within that stream.

Class' relationships. *RingH* models the evolution of inheritance relationships at two levels: (a) a class inherits from a class, and (b) a class is directly inherited by subclasses. Both are modeled with the classes RGSuperInheritanceHistory and RGSubInheritancesHistory respectively. For example, if we have the classes Animal, Monkey, Capuchin and Mandrill, where Animal is the superclass of Monkey, and Monkey is the superclass of Capuchin and Mandrill, then three RGSuperInheritanceHistory objects exist: between Animal and Monkey, Monkey and Capuchin, and Monkey and Mandrill; and two RGSubInheritancesHistory objects exist: between Animal and a collection consisting of Monkey, and Monkey and a collection consisting of Capuchin and Mandrill.

Note that the second relationship can be inferred from the first, however, we determined that such relationship can ease and speed up the querying of hierarchical information in large models. Instead of performing a lookup every time that we need to identify the direct subclasses of a class, the history meta-model provides direct access to this information. We omit RGSuperInheritanceHistory objects from examples shown later to make shorter the representation of history and changes.

Method's relationships. The evolution of method calls, class references and variable accesses is modeled with the classes RGInvocationHistory, RGReferenceHistory, and RGAccessHistory respectively. These relationships are obtained from a method's source code and provide fine-grained information regarding the *history* (as discussed in Section 3.4) that can be used to answer questions such as "What are the current message calls by this method in a particular version?" or "What were the senders of this method in a particular version?" (T_{18} and T_{16} from the catalogue - Section 3.3.2). Additionally, method's relationships cover the majority of possible dependencies between changes within a stream of changes. For example a method depends on the methods being invoked (method calls) and on the classes being referred (class references).

Method's Relationships in a Dynamic Context. Smalltalk is a dynamically typed oriented-oriented programming language. Therefore, in the absence of type information and in the presence of method polymorphism, we need to provide very fine-grained information about method calls. This enables us to be more accurate finding which methods are being called (or invoked) by another method. For example, if method foo calls method bar, we can look for the bar methods that potentially will receive the call (*i.e.,* dynamic dispatch). We refer to that potential set of methods as a *candidate set*.

Figure 6.6 also shows four subclasses of RGInvocationHistory that *RingH* provides to model the evolution of method calls. These classes reflect how method lookup works.

- RGSelfInvocationHistory partially identifies the receiver of a call. This corresponds to the *self* calls within a method m. All candidates for that call need to be situated in the entire inheritance tree of the class in which m is defined. For example, method foo has a self call to method bar, and it is defined in class B. The candidate set is composed of the method bar defined in class B, in its superclasses, and its subclasses.

- RGSuperInvocationHistory identifies the receiver of a call. This corresponds to the *super* calls within a method m. A super call is bound statically. This means that, by analyzing the code, we can determine which method will be invoked. Concretely, this is the method with the correct selector implemented by the direct superclass of the class that defines m. If this superclass does not implement the selector, it is the implementation at the next level higher up in the inheritance tree, and so on. For example, method foo has a super call to method bar, and it is defined in class B. The candidate set consists of the method bar defined in a direct or indirect superclass of B (the superclass that implements a selector bar as it is close to B in the inheritance tree).

- RGStaticInvocationHistory knows exactly which is the receiver of a call. This comes from class references. For example, method foo includes D bar – a reference to class D and a call to method bar. The candidate set is composed of a unique method (bar defined in the metaclass of D).

- RGUnknownInvocationHistory does not know which is the receiver of a call. For example, method foo has a call to method bar. The candidate set consists of every existing method bar.

Note that by defining the different kinds of method calls, we are able to decrease the amount of false positives that may exist in a *candidate set*. Hence, it decreases the number of false positives of dependencies between changes within a stream as well.

Representation of Systems' Histories. We have described mainly the classes that model the evolution of program entities and relationships. Here, we illustrate how the history of a system is defined by means of *RingH*.

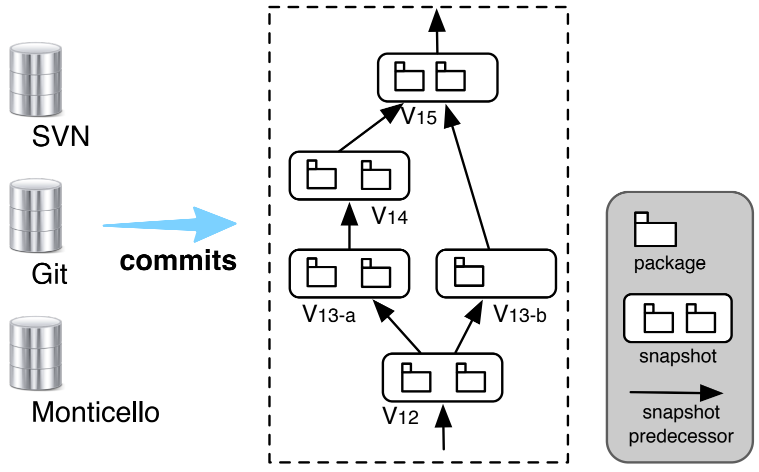


Figure 6.7: *RingH: representing system's histories.*

Figure 6.7 shows an overview of a resulting system's history modeled with *RingH*. Note that the history of a system – in terms of source code – is retrieved from a versioning repository (*e.g.,* Monticello). That history is defined in terms of a graph of snapshots.

The definition of *snapshots* was introduced in Section 3.2. A snapshot defines the complete view of a system at a given time. It contains the program entities as well as the relationships between such entities. Therefore, a snapshot represents the context in which a program entity needs to be resolved, and it could be conceptually considered as a *version* of the system. *RingH* models snapshots by means of the class RGSnapshot.

RGSnapshot objects are treated as any historical object: they can have predecessors and successors. The arrowed edges shown in Figure 6.7 correspond to the links between snapshots and their predecessor. Note that by means of this we can assemble the whole history and we are able to model branches. For example, the snapshot V_{12} is the predecessor of snapshots V_{13-a} and V_{13-b}, which at the same time indicates branching. Both branches were later merged in snapshot V_{15}.

Snapshots originated from commits contained within a versioning repository. In our context, the Monticello version control system is package-based. It lacks explicit commits: instead packages are individually versioned (*i.e., package versions*) containing their whole definitions (*i.e.,* not just changes). Therefore, a commit is determined by grouping package versions that belong together. We use a sliding window technique [Zimmermann 2004a] that considers that several packages belong to the same commit if they are committed by the same committer within a time interval of 5 minutes.

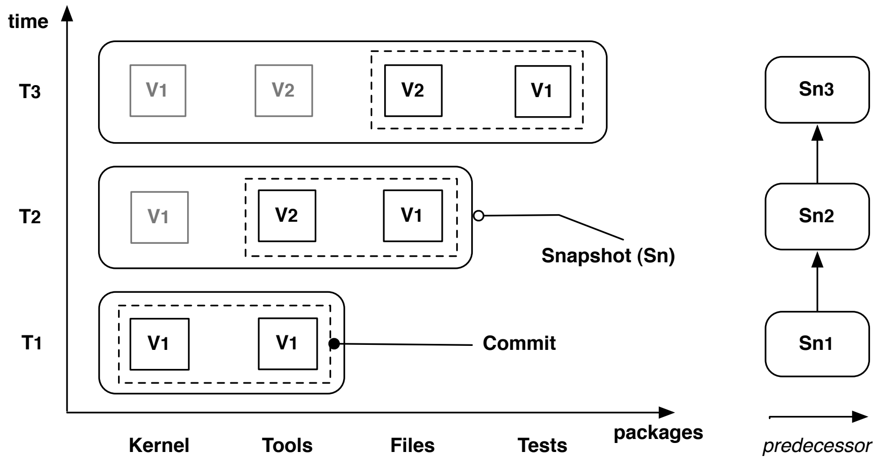


Figure 6.8: *Commits and snapshots.*

The difference between a commit and a snapshot is that a commit refers to a set of changes (in Monticello to the packages that were changed), whereas a snapshot refers to the complete system. In Figure 6.8, we illustrate how commits and snapshots are established by simulating a scenario in the context of Monticello.

Along the *x* axis we show the different packages that are contained within a Monticello repository. The *y* axis represents the various sliding windows[2] [Zimmermann 2004a](T_n) at which a commit occurred.

At T_1 the first version V_1 of packages *Kernel* and *Tools* were published. Both versions represent the first commit and also the first snapshot S_1. Next, at T_2 a new version was committed of the previously existing package *Tools* (V_2), and a newly created package *Files* (V_1) was added to the system. The changes made to these two packages correspond to the second commit and hence also snapshot S_2. Note that the second snapshot S_2 also includes the package *Kernel* (V_1) that was already

[2]A sliding window is a time period that stretches back in time from the present.

present in the repository. Finally, at T_3 a third commit was published containing a new version of the package *Files* (V_2) and the first version of the package *Tests* (V_1). The third snapshot S_3 includes both packages and also includes the unchanged packages *Kernel* (V_1) and *Tools* (V_2) as they were part of the system in that sliding window.

Our example applies to Monticello, but the same approach can be used to represent the history of systems stored by other version control systems such as Git, Subversion, and so on. In fact, for such sources it is rather simple to model the history because they keep explicit commits.

As we mentioned before, snapshots represent a complete view of the system. Based on the requirement *"model only changed program entities and relationships"*, the evolution of program entities and relationships only takes into account the presence of changes (*i.e.,* when they were introduced, modified and removed). Therefore, program entities and relationships need to be shared in the snapshots in which they have not changed. For example, if class Zoo was added in snapshot S_1 and modified in snapshot S_4, the first representation of class Zoo is shared by the snapshots S_2 and S_3.

Querying information in a history that shares unchanged entities between snapshots requires a means to explicitly indicate which is the context (*i.e.,* snapshot) to resolve queries. *RingH* includes a special class RGHistoryWrapper to support queries between snapshots and optimize navigation. A wrapper links a historical object (except snapshots) together with the snapshot in which that object exists (*i.e.,* its context). For example, if Zoo is queried in S_4 the result is a wrapped object of the second Zoo object and the snapshot S_4 (because it exists in S_4); any message that is sent to Zoo is then resolved in the context of S_4.

6.2.3 Importing the History of a System

We propose an approach that imports the history of a system stored into a version control system's repository. Currently, *RingH* is developed in Pharo and we aim to process the history of Smalltalk projects stored in Monticello repositories. However, our architecture can be applied to other programming languages such as Java, and to version control systems such as Git or SVN with some engineering effort.

The history of a system is represented by a *graph of snapshots* as shown in Figure 6.7. Note that to support other programing languages the history meta-model might need to be extended in order to define specific languages constructs (*e.g.,* interfaces in Java) that are not present in Smalltalk. Supporting other version control systems depends on providing the importers for these respective systems.

In the following, we present a high-level overview of the algorithm behind our history importer for Smalltalk projects.

Algorithm: Importing the History

1. **Find all versions to import** (versions in the project's repository and external repositories if merges happened)
    (a) **Calculate a package graph** containing all the versions
    (b) **Calculate a commit graph** based on the package graph
    (c) **Calculate a snapshot graph** based on the package graph and commit graph

   (d) **Apply a topological sort** to the snapshot graph to order the nodes based on how the versions were published into the repositories

2. For each node in the snapshot graph:
   (a) **Set the current model** (a RGSnapshot object where the import happens)
      - If the node has no predecessors directly instantiate the class RGSnapshot
      - If the node has predecessors, make a descendant of the first of its predecessors and merge that descendant with the other predecessors

   (b) For each version associated to the node:
      - **Find the packages in the version**
        - For each package:
          - **Import the package** into the current model
          - For each class contained in the package:
            - **Import the class** into the current model
            - **Import the attributes of the class** into the current model
            - **Import the methods of the class** into the current model
          - For each trait contained in the package:
            - **Import the trait** into the current model
            - **Import the methods of the trait** into the current model

   (c) From the imported methods:
      - **Import class extensions** into the current model

   (d) From the imported classes and methods:
      - **Import class inheritances** into the current model
      - **Import method invocations** into the current model
      - **Import class references** into the current model
      - **Import attribute accesses** into the current model

6.2.4 Metrics and Memory Footprint

We imported and generated *RingH* models of the history of three Smalltalk projects as a means to illustrate several metrics and memory footprint of the our importer and *RingH*.

- **Ring.** We selected the history of *Ring* and *RingH* contained in the repository located at http://www.squeaksource.com/Ring

- **Monticello.** We selected the history of the Monticello version control system in Squeak contained in the repository located at http://source.squeak.org/trunk

- **Fuel**[3] is a general-purpose fast object serialization framework developed in Pharo. We selected the history of the core of Fuel contained in the repository located at http://www.squeaksource.com/Fuel

	Monticello	Ring	Fuel
Number of versions imported	196	290	554
Number of packages imported	392	5777	5768
Number of classes imported	47702	60376	132808
Number of methods imported	267070	611205	505133
Lines of code imported	1509024	3110084	3148265
Number of classes in representation	518	1977	1585
Number of methods in representation	3002	13665	5976
Total amount of memory (Mb)	101	163	216

Table 6.1: *Metrics and memory footprint and of three RingH models.*

Table 6.1 shows the measurements of imported entities, lines of code and classes/methods represented in the model, and the memory footprint obtained from the generation of these three *RingH* models.

We have divided the metrics in two groups. The first group of metrics represents the number of versions, packages, classes, methods and lines of code that were imported from a repository. This means the amount of code and the number of entities that occur in the history of a project that was imported. Note that the histories of the three projects are quite large. For example, the smallest history corresponding to Monticello consists of 1509 KLOC, and the largest history corresponding to Fuel consists of 31483 KLOC. These projects are not trivial and their source code histories are made up of a huge number of entities.

The second group of metrics represents the number of classes and methods that were actually represented (created) within the *RingH* model. This means for example, for the Fuel project, that out of 132808 classes existing in its source code history (*i.e., number of classes imported*) only 1585 classes have to be represented in the model (*i.e., number of classes in representation*). This number corresponds to all the changes made to classes in the source code history. Remember that the version control system used by Pharo or Squeak – namely Monticello – versions packages containing all their definitions (*i.e.,* classes and methods) and not only changes. Our history representation only keeps track of entities that actually changed within the history. This is one of the requirements established for *RingH* in Section 6.2.1 as a means to optimize memory consumption and minimize object creation time. The same difference exists for methods. While we imported 505133 methods from the source code history of Fuel, within the *RingH* model we only needed to represent 5976 methods. This number corresponds to the number of changed methods in the source code, which is significantly smaller than the total number of methods imported 505133 (1.18%).

The last row presented in the table shows the memory footprint of the history models. In the case of Fuel, the entire history model only takes up 216 Mb of memory. As a result, we are able to load the entire history of Fuel in memory and analyze it.

Regarding the time that was needed to import and create the model, we report that all the imports ran under 10 minutes.

[3]Fuel: http://rmod.lille.inria.fr/web/pier/software/Fuel.

6.2.5 Creating Objects in the History Model

We have already described *RingH* and the underlying algorithm of the history importer. Our history meta-model was conceived in such a way as to minimize memory consumption and object creation time, and to easily query its information in order to support the representation of large systems with large histories. Each program entity and relationship uses a unique identifier that remains stable over evolution. Moreover, we only fully represent program entities and relationships when they actually changed with respect to a previous version. This implies that snapshots may share unchanged objects. Note that both are requirements established for *RingH* as explained in Section 6.2.1, and they are the optimization of similar ideas proposed by Orion as explained in Section 6.1.2.

These requirements are key points in our history meta-model to assist the integration process of large systems in the presence of multiple branches. Snapshots share entities that did not change between versions. They keep track of the unique identifiers of objects along with the objects themselves representing program entities or relationships that existed in the system at a given time. The rest of the objects in the model (*e.g.,* packages, classes, methods) only keep pointers to the identifiers of other objects related to them (*e.g.,* a package only keeps the identifiers of the classes it contains).

In the following we show a basic scenario to illustrate how the program entities and relationships are created, and how the lookup of entities occurs upon evolution. Concretely, consider the scenario where a system is composed of one package (Model), and in the history exist two versions of this package. In the initial version, the package Model contains three classes (Zoo, Animal, Lion). The class Zoo defines the attribute animals and two methods (addAnimals, feedAnimals), and the class Animal defines the method eat. The class Lion is a subclass of Animal but it does not redefine any behavior. In the second version, the following changes occurred: the class Lion was removed and the class Animal introduced the attribute name.

Figure 6.9 shows the graph of snapshots representing the two versions of this system, and how the entities are created along with their respective identifiers. Note that for each version (*i.e.,* commit) a snapshot was created. We display snapshots as rounded rectangles, and their predecessor links by means of an arrowed edge. An edge from $Snapshot_1$ to $Snapshot_2$ denotes that $Snapshot_1$ is the predecessor of $Snapshot_2$.

Within a snapshot we have included a UML-like model of the system in the top left side, the entity table (equivalent to a v-table) mapping the identifiers to the entities that the snapshot can resolve in the top right side (*e.g.,* ID_2 corresponds to the id of the class Zoo), and how the entities interact with other entities shown in the bottom side. The UML-like model shows a simple view of how we map the entities and relationships onto the objects using $RingH$.

$Snapshot_1$ contains an entity table with the identifiers and objects existing at that time. This table contains 8 definitions (from identifiers ID_1 to ID_8) representing the program entities (package, classes, attribute and methods), and 4 definitions (from identifiers ID_9 to ID_{12}) representing relationships (attribute access, class reference, method invocation, and class inheritance). Note that all the objects displayed at the bottom have only pointers to the identifiers of other objects, for example class Zoo has a pointer to the identifier ID_5 of its attribute animals and two pointers to the identifiers ID_6 and ID_7 of its methods addAnimals and feedAnimals respectively.

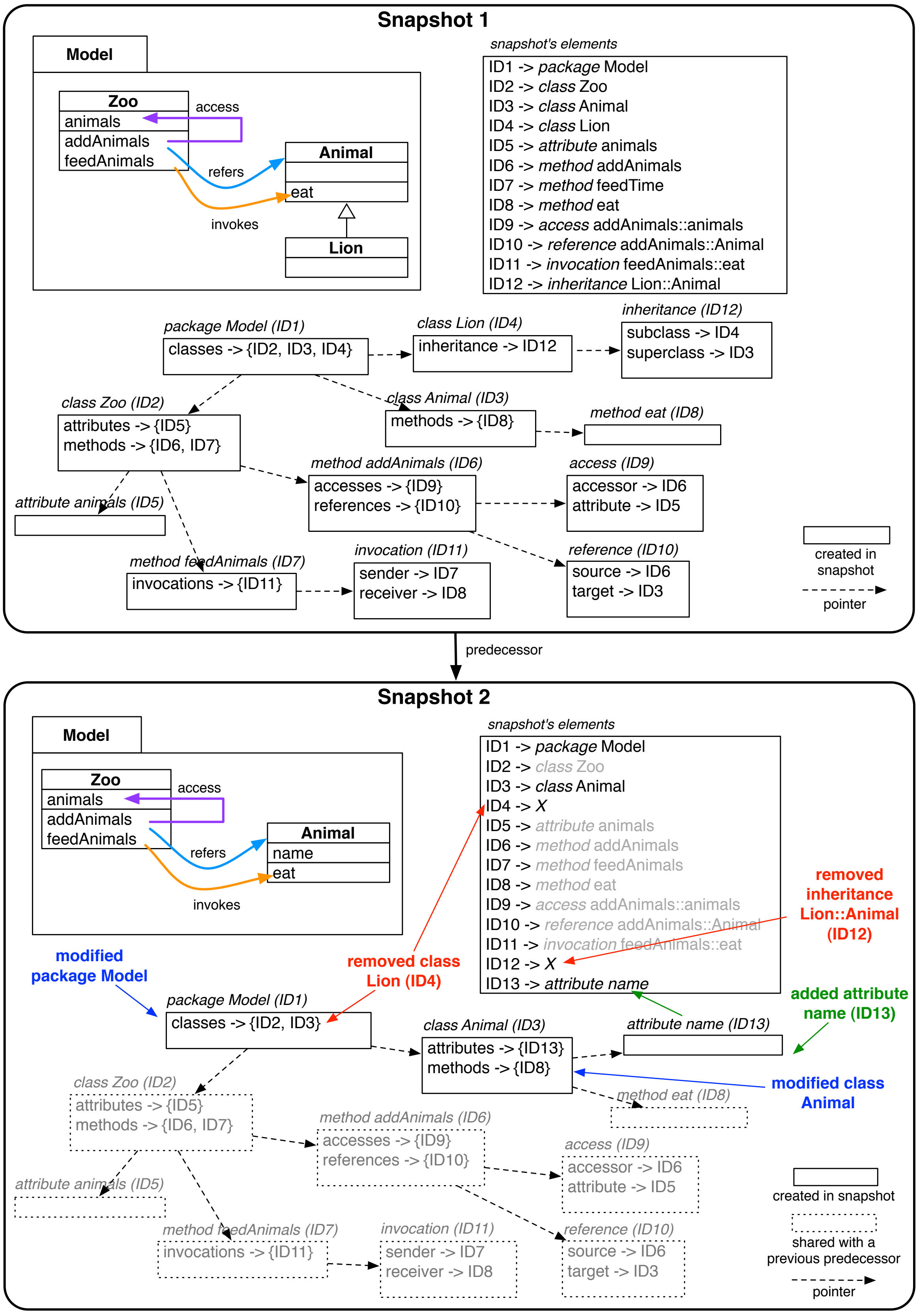


Figure 6.9: *Creation of program entities and relationships in the history.*

$Snapshot_2$ reflects the changes between both versions: the removal of the class Lion and the addition of the attribute name in the class Animal. $Snapshot_2$'s table contains the definitions of the objects in the history as follows:

- The object associated with ID_4 was replaced by a RGRemovedHistory object representing the removal of the class Lion (we show the removal as an X to make it simple).
- The object associated with ID_{12} was replaced by a RGRemovedHistory object representing the removal of the class inheritance relationship between the removed class Lion and its superclass Animal.
- A new definition was added (ID_{13}) to represent the attribute name introduced in class Animal.
- The objects associated to ID_1 (package Model) and to ID_3 (class Animal) were replaced by new objects created in this snapshot. This is due to the class removal and attribute addition that affected both respectively.
- The rest of the definitions are shared with its predecessor $Snapshot_1$. Note that $Snapshot_2$ tracks the identifiers of the objects that did not change, and shares these objects with $Snapshot_1$ (shown in grey).

The fact that the class Lion was removed implied that the package Model no longer contains a pointer to ID_4. Therefore, a new version of Model was created in $Snapshot_2$. Something similar happened to the class Animal, as it introduced the attribute name, a new version of this object has to be created in $Snapshot_2$ to include a pointer to the newly created object representing the attribute name with identifier ID_{13}.

Finally, note at the bottom in $Snapshot_2$ that we display the objects that did not change using grey dashed rectangles. They are the shared objects with its predecessor $Snapshot_1$.

6.2.6 Querying the History Model

Searching for information within a model (*i.e., querying* that model) allows tools to navigate between entities of the model. Basic queries may represent questions like: "What are the methods of a class?", "What are the invocations of a method?", "What is the superclass of a class?", "What are the classes contained in a package?", and so on. Complex queries are composed of basic queries, for example: "What are the superclasses of the classes contained in package System that define the method test, which in turns invokes the method run?"

Querying the history represented using *RingH* implies taking into account the fact that snapshots may share entities. For example, starting from a given snapshot (S_5), a query may run on shared entities from older snapshots (S_2, S_1). The results must always be interpreted in the context of the given snapshot (S_5), as the shared entities may point to entities that have since changed. Therefore, to run a query we need to know the context (*i.e.,* snapshot) in which such a query will be interpreted.

The challenge of running queries over shared entities given a current snapshot is summarized as follows:

- Queries retrieve entities which may or may not reside in a parent snapshot.
- *RingH* should resolve each retrieved entity in the correct snapshot.

Resolving entities in the history can be compared to late binding. However, in *RingH* there is no look-up through predecessor snapshots, but a direct access through the entity table (equivalent to a method table or v-table) of the snapshot that keeps track of the identifiers and objects. Where that snapshot is the context to resolve a query.

Using the example shown in Figure 6.9, we illustrate the following queries written in Smalltalk. We use the notation *query* ⇒ *results*.

1. S_2 packages ⇒ { Model }
   This is a basic query that returns the packages in the context of S_2. Note that Model was modified in S_2, therefore, it is not a shared entity with S_1.

2. (S_1 packageNamed: 'Model') classes ⇒ { Zoo, Animal, Lion }
   This query returns the classes of the package Model in the context of S_1. This is a composed query that (1) resolves the package Model, and (2) resolves the classes of Model. Note that the package and the classes all reside in S_1.

3. (S_2 packageNamed: 'Model') classes ⇒ { Zoo, Animal }
   This query returns the classes of the package Model in the context of S_2. Class Zoo exists in S_2 but resides in S_1 (as this entity did not change). This is an example of shared entities between snapshots. Zoo is reachable in S_2 because its table has a key-value pair pointing to the most recent entity with respect to S_2.

4. (((S_2 packageNamed: 'Model') classNamed: 'Zoo') methodNamed: 'addAnimals') references ⇒ { Animal }
   This query returns the class references of the method Zoo»addAnimals contained in package Model in the context of S_2. This composed query (1) resolves the package Model that resides in S_2, (2) resolves the class Zoo that exist in S_2 but resides in S_1, (3) resolves the method addAnimals that exist in S_2 but resides in S_1, and (4) resolves the class reference to Animal that resides in S_2. Note that the query did not return the class Animal residing in S_1 even though the method requesting the references in addAnimals resides in S_1. Instead, the class Animal was interpreted in the context S_2 where the query is running.

6.3 RingC: a Change and Dependency Model and Analyses

In Chapter 3 we introduced definitions such as *changes*, *deltas* and *dependencies*. In this section, we present our meta-model, *RingC*, that allows us to model such definitions, and create a change-based representation of a system's history. By means of a change model of the history, we can analyze and characterize streams of changes to assist the integration process across branches.

6.3.1 Architecture of RingC

RingC is a change and dependency meta-model built on top of the *Ring* source code meta-model described in Chapter 4, and it is an application of the *RingH* history meta-model described in Section 6.2. Within *RingC* we can represent changes, deltas, change dependencies, and external dependencies as first-class entities. Figure 6.10 shows an overview of these definitions of the meta-model.

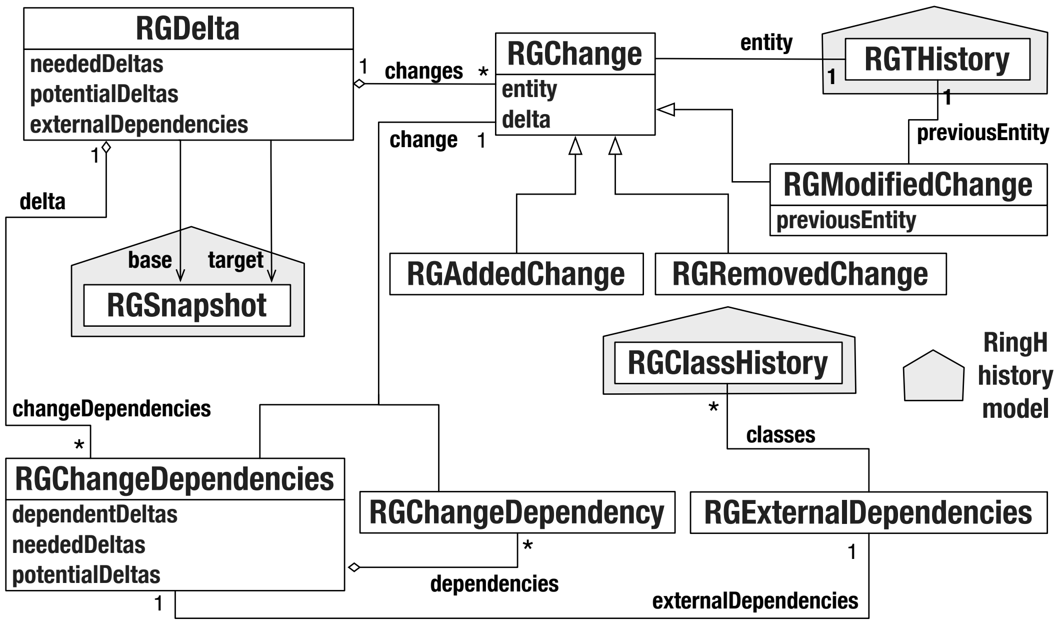


Figure 6.10: *RingC change and dependency meta-model - Key classes.*

Changes. With *RingC*, we represent changes as first-class entities. Having this representation of changes enable us to perform a straightforward dependency analysis. Changes are extracted from the history representation of a system (*i.e.,* from the graph of snapshots modeled with *RingH*).

Three kinds of changes can be modeled: additions, modifications, and removals of program entities and relationships. *RingC* provides the classes RGAddedChange, RGModifiedChange, and RGRemovedChange to represent additions, modifications and removals respectively. Note that they all inherit from RGChange.

A change object knows the program entity or relationship that was changed upon evolution. This creates a link between the change-based representation of the evolution of a system to the history-based representation of such evolution. Figure 6.10 shows how the change model explicitly refers to the history model (shown with a grey background).

Note that a change object due to a modification, not only knows the program entity or relationship that changed, but also its previous program entity or relationship.

Deltas. A delta defines the differences between two snapshots in terms of a set of changes. Within *RingC*, we provide the RGDelta class to model deltas. A delta knows its predecessor(s) delta and successor(s) delta, where they represent the previous set of changes and the subsequent set of changes that happened regarding that delta. This follows the same approach as for snapshots, and serves to model a stream of changes as a graph of deltas as shown later in Figure 6.14. A delta contains a set of RGChange objects and by means of the dependency analysis (explained in Section 6.4.1) a delta can identify the set of dependencies of its changes and its dependencies to other deltas within the stream.

We illustrate changes and deltas using the same scenario introduced in Section 6.2.5. It was originally used to illustrate how objects are created within a history model (shown in Figure 6.9). This scenario presents two snapshots of a simple system consisting of the package Model and two classes Animal and Lion. The second snapshot changed with respect to the first snapshot by removing the class Lion and adding the attribute name to Animal.

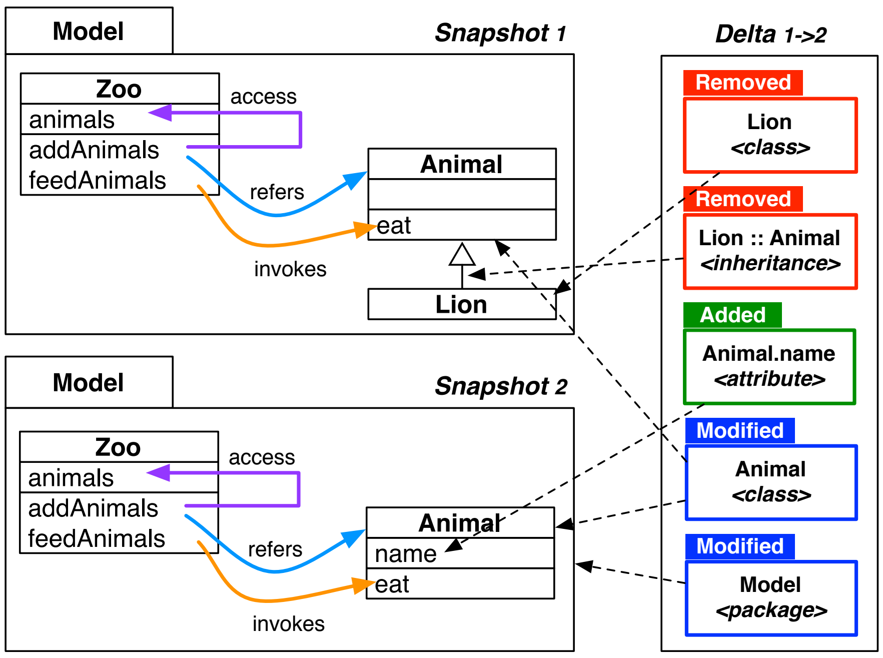


Figure 6.11: *Deltas and changes.*

Figure 6.11 shows how a delta expresses the differences between two snapshots. Note on the left side, that we show the UML-like models representing both snapshots instead of the resulting history model to facilitate the discussion. Here, we do not provide details of how changes and deltas are calculated. That is described in Section 6.4.1.

On the right side, we show the delta expressing the changes between $Snapshot_1$ and $Snapshot_2$. Additions, removals and modifications are shown in green, red and blue respectively. The removal of class Lion is represented as (1) removal of the class itself, (2) removal of the class inheritance (Lion inherited from Animal), and (3) modification of the package Model (Model in $Snapshot_2$ does not refer to Lion). The addition of the attribute name to class Animal is represented as (1) addition of attribute, and (2) modification of the class Animal (Animal in $Snapshot_2$ refers to name). Each change knows the respective object(s) in the snapshots (shown by means of the arrowed dashed edges).

Change Dependencies. A *change dependency* captures the fact that a given change potentially depends on another change. In *RingC*, the RGChangeDependency class models a dependency between two changes. This dependency can exist between changes within the same delta or between changes in different deltas. Our approach characterizes a change dependency based on the locality and size of potential changes that can satisfy this dependency.

Delta Dependencies. Our approach makes use of the notion of dependencies between deltas to characterize deltas within a stream. A *delta dependency* expresses a dependency between two deltas based on the *change dependencies* between changes. Concretely, a delta D_n depends on delta D_m if a change CH_y in D_n depends on at least one change CH_x in D_m. Delta dependencies are not modeled as a first-class entity, instead a delta knows the deltas on which it (potentially) depends.

In Figure 6.12, we illustrate both kinds of dependencies: change dependencies and delta dependencies. *Change dependencies* are shown using directed dashed edges and *delta dependencies* are

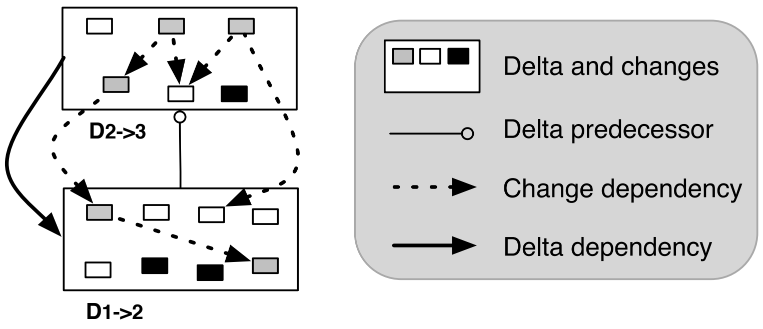


Figure 6.12: *Change and delta dependencies.*

shown using directed edges. This example contains two deltas. The delta $D_{1\rightarrow 2}$ and its successor delta $D_{2\rightarrow 3}$. Delta $D_{2\rightarrow 3}$ contains 2 changes that depend on 2 other changes of the same delta, and 2 changes that depend on changes contained in the predecessor delta $D_{1\rightarrow 2}$. Therefore the delta $D_{2\rightarrow 3}$ has a *delta dependency* on its predecessor $D_{1\rightarrow 2}$.

External Dependencies. A change within a stream may depend on a class defined in external components. For example, the change to method foo introduced a reference to the class B. However, the class B does not exist within the stream. In this case, we say that the modification to method foo has an *external dependency* to class B.

RingH models external classes as stub classes, therefore they are also part of the history-based representation of a system. Such information is then used to establish potential external dependencies present in the change-based representation.

Within *RingC*, we model external dependencies of a change by means of the class RGExternalDependencies. An RGExternalDependencies object knows the stub classes on which a change depends.

6.3.2 Deriving the Change Model from the History Model

The characterization of streams of changes relies on the dependencies between changes and between deltas themselves within a stream. To enable de calculation of dependencies we require a representation of the history in terms of deltas and changes. In other words, we need a change-based representation of the history.

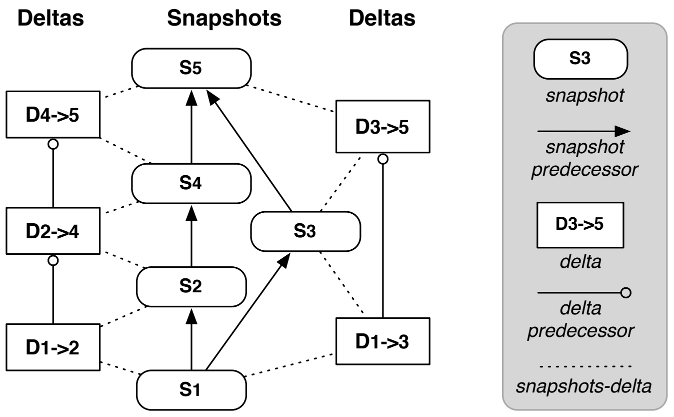


Figure 6.13: *Graphs of snapshots and deltas.*

From the history (*i.e.,* a graph of snapshots), we derive a change-based representation (*i.e.,* a graph of deltas). Figure 6.13 illustrates how a graph of snapshots can be used to obtain a graph of deltas. In Section 6.4 we describe the calculation of the change-based representation.

The graph of snapshots appears in the middle of the graph of deltas. Snapshots are shown with rounded rectangular shapes, while deltas are shown with rectangular shapes. A delta knows the base and target snapshots from which it was defined (shown by means of the dashed lines). For example, $D_{1\rightarrow2}$ defines the differences between S_1 and S_2.

As explained before, a delta also knows its previous delta (predecessor) and its subsequent delta (successor). For example, $D_{1\rightarrow2}$ is the predecessor of $D_{2\rightarrow4}$, and $D_{2\rightarrow4}$ is the successor of $D_{1\rightarrow2}$. These relationships are denoted by means of a line with a circle towards the successor. They allow to assemble the graph of deltas.

6.4 Calculating Deltas and Dependencies from the Stream

In this section we describe the analyses that allow the calculation of deltas and dependencies used to *characterize streams of changes*. We explain this characterization in Chapter 7.

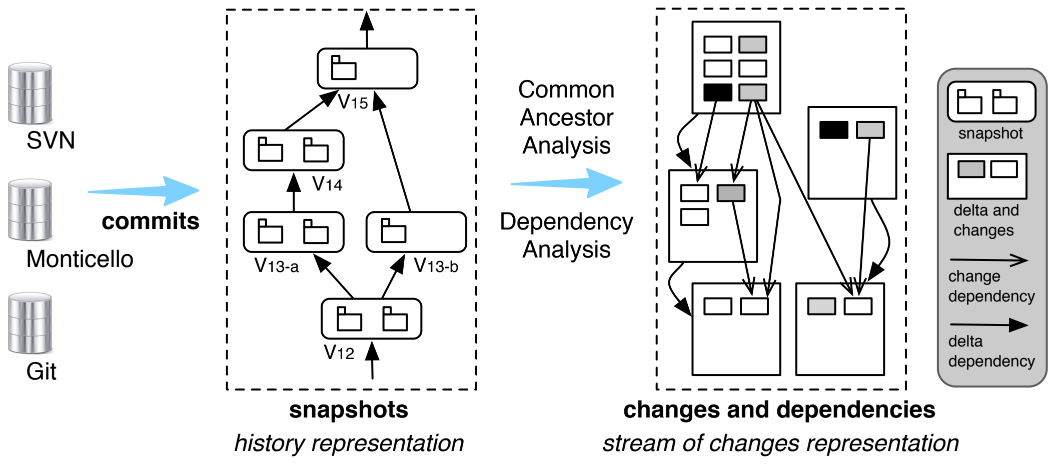


Figure 6.14: *Approach architectural overview.*

We propose an approach that enables the characterization of successive sets of changes (*i.e.,* deltas) and their dependencies within a stream of changes. Figure 6.14 shows an overview of the architecture of this approach. Note that the first part of the figure (from left to right) corresponds to the representation of a system's history by means of *RingH* as described in Section 6.2.2 (also shown in Figure 6.7). Therefore, the history is the input data for the definition of changes and dependencies. This process is based on the common ancestor analysis and dependency analysis (shown in the middle), and results in a representation of the history in terms of deltas, and change and delta dependencies by means of *RingC* (shown on the right).

More concretely, from the graph of snapshots representing a system's history, our approach computes a graph of deltas and dependencies between the changes of each delta and between deltas themselves.

In the rest of this section, we describe our algorithm to calculate deltas and the dependencies between deltas.

6.4.1 Delta Mechanism

Our algorithm to calculate deltas and the dependencies between deltas takes as input a graph of snapshots and computes a change-based representation of this graph of snapshots, along with the dependencies between and inside deltas.

We divide our algorithm into two different stages: (1) calculating deltas, and (2) finding dependencies. First we introduce the stage dedicated to calculate deltas and then we proceed to explain the main steps of the process.

Stage 1: Calculating Deltas

1. Find all root snapshots (snapshots with no predecessors)
2. For each root snapshot:
   (a) **Calculate a root delta** containing additions of the snapshot's elements
   (b) Traverse the graph of snapshots from the root to its most recent successor(s)
      - **Calculate a delta** for each pair (predecessor, successor) containing their differences
3. Assemble a graph of deltas by finding the predecessors of each delta
4. Retrieve all merged snapshots (snapshots with more than one predecessor)
5. For each merged snapshot, **refine related deltas** by taking the common ancestor of the snapshots that originate the merge into account

Calculating Root Deltas

We consider all snapshots that do not have predecessors to be root snapshots. Earlier, we have defined a delta as the representation of the differences between two snapshots. As *root snapshots* do not have any predecessors, we introduce here the notion of *root deltas*. We compute a root delta by creating RGAddedChange objects for each of the program entities and relationships defined in the root snapshot.

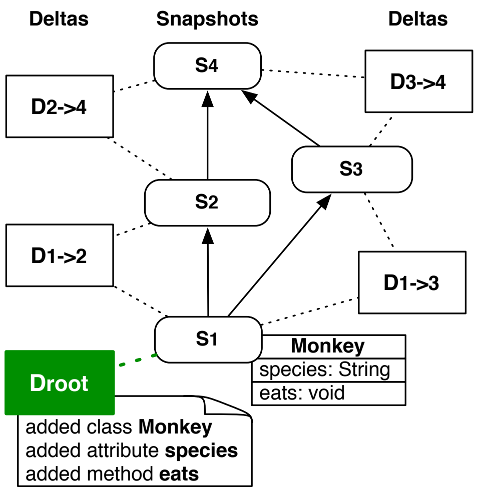


Figure 6.15: *Root delta: completing a change-based representation of a stream of changes.*

Figure 6.15 shows a graph of snapshots (in the middle) containing one root snapshot S_1. The root snapshot contains a single class Monkey and two methods breed and eat. Therefore, the *root delta* D_{root} contains *added changes* for each of these program entities: an *added class* for Monkey, an *added attribute* for species and an *added method* for eat (shown with a UML-like diagram). Note that from the method's body, other *added changes* may be created to represent the method's relationships (method calls, class references and attribute access). We omit the relationships here to avoid cluttering the figure.

Calculating Deltas

A delta is computed by extracting the differences between a pair of snapshots (predecessor, successor). The differences are then reified as changes and represented as additions, modifications and removals using the *RingC* change and dependency meta-model shown in Figure 6.10.

We consider that an entity has been modified when its *definition* has been changed:

- *Package.* If a new subpackage was added or an existing subpackage was removed.
- *Class.* If its name, superclass, variables or comment changed.
- *Method.* If its source code (including signature) or its protocol[4] changed.

The deltas are calculated by traversing the graph of snapshots, starting with the root snapshots until we reach snapshots that do not have any successors.

We illustrate this process in Figure 6.15. The root snapshot S_1 has two successors S_2 and S_3 (which reflects branching). For each pair of snapshots (predecessor, successor) a delta is calculated. When applied to our example this results in two deltas: delta $D_{1 \to 2}$ for the pair (S_1, S_2) and delta $D_{1 \to 3}$ for the pair (S_1, S_3). After processing snapshot S_1, we continue traversing the graph via the successors of S_2 and S_3. The snapshot S_2 has a successor S_4 which results in the delta $D_{2 \to 4}$. Finally, the snapshot S_3 has a successor S_4 which results on the delta $D_{3 \to 4}$.

After traversing the whole graph of snapshots and computing deltas, we assign the predecessors and successors of each delta based on the predecessors and successors of the snapshots that generated such delta. The root delta D_{root} has two successors $D_{1 \to 2}$ and $D_{1 \to 3}$. $D_{1 \to 2}$ has as predecessor D_{root} and as successor $D_{2 \to 4}$, whereas, $D_{1 \to 3}$ has as predecessor D_{root} and as successor $D_{3 \to 4}$.

Refining Deltas in the Presence of a Merge

Up until now, our calculation of deltas does not take into account that a snapshot might be the result of merging two snapshots. One of the requirements for *RingH* is to *represent the history made of multiple branches and merges*, therefore our change model *RingC* also takes branches and merges into account. Branches play an important role for our contribution considering that our goal is to assist the integration of changes between branches, *i.e.,* support cherry picking. In particular we aim at the characterization of streams of changes which due to a collaborative development process it may be composed of multiple branches.

Based on Figure 6.15, the snapshot S_4 is a merge between the snapshots S_2 and S_3, or in other words a merge of two branches in the history. The snapshots S_2 and S_3 correspond to the predecessors

[4]Methods in Smalltalk are classified in protocols.

of S_4. For each of them, we have calculated a separate delta resulting in $D_{2\to4}$ and $D_{3\to4}$. However, due to the merge, each of these deltas might be "polluted" with a number of changes that might have occurred in the other branch. As we are only interested in the changes that *contribute* to a merged snapshot, we perform a post-processing step on all deltas that are associated with a merged snapshot to prune away changes that do not contribute to the result of the merge. To this end, we propose a technique similar to three-way merging algorithms [Lindhom 2001].

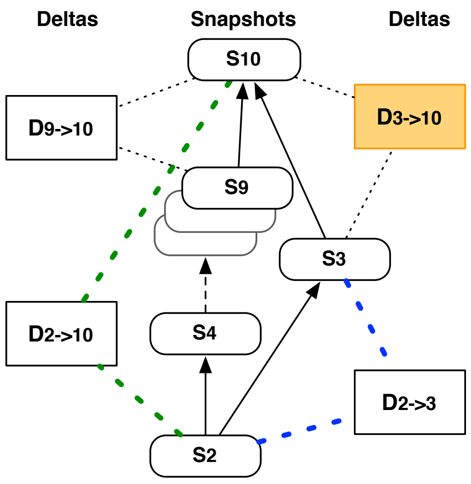


Figure 6.16: *Deltas in the presence of merge: taking common ancestor S_2 into account.*

We illustrate our technique by means of a slightly more complex scenario, as shown in Figure 6.16. In this scenario, we have a snapshot S_{10} that is the result of merging S_3 and S_9. Note that both predecessors have a common ancestor – namely S_2 (shown at the bottom). Assuming that S_3 is closer to the common ancestor S_2 than S_9, the delta $D_{3\to10}$ (shown in orange) potentially contains a number of changes that may be present in the other branch from the snapshots S_4 to S_9. Note that changes from S_9 are merged in S_{10}. Therefore, such changes are unrelated to the changes of S_3 that actually contribute to the merged snapshot S_{10}.

In other words, we are interested in all the changes that have occurred between snapshots S_2 and S_3 (indicated with the blue dashed lines) together with all the changes between S_3 and S_{10}, minus the changes happening in the other branch from S_2 to S_{10}. If we generalize this, we obtain:

$$D_{op\to m} - D_{ca\to m} + D_{ca\to p}$$

where *op* is the oldest predecessor, *m* the merge, *ca* the common ancestor, and *p* the predecessor.

Applying this formula to refine $D_{3\to10}$, we see that the delta is obtained by computing $D_{3\to10}$ (original delta) - $D_{2\to10}$ (indicated with the green dashed lines) + $D_{2\to3}$ (indicated with the blue dashed lines).

6.4.2 Dependency Mechanism

Once changes and deltas are established, the second part of the process is to calculate the dependencies between changes and deltas. We introduce the second stage of our algorithm and then we proceed to explain the main steps of the process.

Stage 2: Finding dependencies
For each delta:

1. **Filter the changes within the delta**: only select those that *may* depend on another change
   (a) Include additions and modifications of classes and methods
   (b) Exclude modifications of methods that do not introduce new method calls or class references
2. For each change that may depend on other changes, **determine its dependencies**:
   - A change to a class depends on:
     - The most recent change to its superclass
     - If such a change does not exist, its superclass is an external reference. Add a dependency to this external reference.
   - A change to a method depends on:
     - The most recent changes to the potentially called methods.
     - The most recent changes to the referred classes.
     - If changes to the referred classes do not exist, the method refers to external classes. Add a dependency to each external reference.
3. Prune **redundant delta dependencies**

Filtering Changes Within a Delta

Not all changes within a delta lead to the introduction of dependencies. As a pre-filtering step, we partition the changes within a delta into two groups: (1) changes that potentially depend on other changes, and (2) changes that do not depend on another change.

For the first group, we only consider additions and modifications of classes and methods that result in the introduction of dependencies. The reason is that we only require dependencies that are needed when integrating changes, and therefore we do not include removals. Furthermore, we exclude modifications to methods that changed their source code without introducing or removing method calls and class references (*e.g.,* when a method only changed comments or variable accesses, or even when lines of code were moved around). These changes do not introduce or remove dependencies to other changes. Changes to classes that did not change the superclass of the class are also filtered out. This considering that *e.g.,* adding an attribute to a class does not introduce any dependency. All other changes within the delta are considered to belong to the second group.

Determining Dependencies

We proceed to determine the dependencies for each change within a delta that was categorized as a change that may potentially depend on other changes. A *change dependency* is a relation between two changes, where both changes can be present in the same delta or in different deltas. Changes to classes and methods can depend on other changes based on the following rules:

- **Class level**: Changes to a class depend on the most recent change to its superclass. The reason is that to integrate a class we also require its superclass.

- **Method level**: Changes to a method depend on:
  - **Change to class references**: The most recent changes to the referred classes.
  - **Change to method calls**: The most recent changes to potentially called methods (*i.e.,* candidate set). The set of potentially called methods is bounded statically taking into account polymorphism.

To minimize false positives in the candidate set of a method call from method m, we identify the potential receiver of the call (*i.e.,* the class that understands a message with the name of the call). This can be (a) a superclass in the inheritance tree of the class defining m, (b) any class in the inheritance tree of the class defining m, (c) the class of a reference, (d) any class implementing a method with the same name of the call. In Section 6.2.2, we described how these method calls are modeled.

To determine the most recent change of an entity, we make use of the graph of deltas that was determined in a previous step of our algorithm.

For example, delta D_m contains a modification to the class Monkey that changed its superclass from Animal to Mammal. Therefore, that change now depends on the *most recent* change within the graph of deltas (and with respect to delta D_m) of the superclass Mammal.

If the superclass Mammal was never modified, its most recent change corresponds to the addition of this class. Note that, if this class was never added in the history under analysis, it is considered to be a dependency on an external entity.

Based on the dependencies between changes, we also compute the dependencies between deltas (*delta dependencies*). We say that delta D_1 is dependent on another delta D_2, if there exists at least one change in D_1 that depends on a change in D_2.

Pruning Redundant Delta Dependencies

Note that our algorithm for calculating delta dependencies can result in redundancies. To illustrate this, consider the left graph of deltas depicted in Figure 6.17, where delta dependencies are indicated by means of a black directed edge.

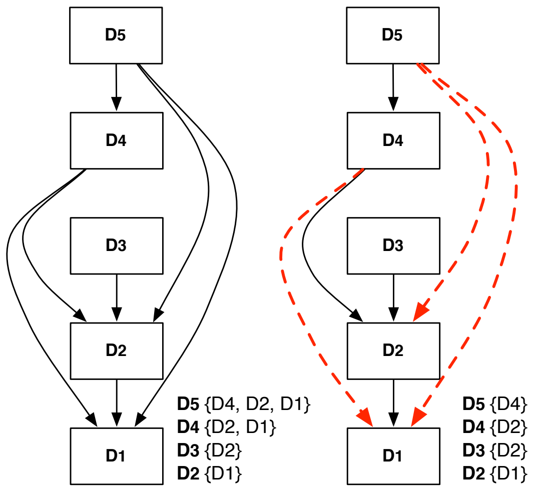


Figure 6.17: *Redundant delta dependencies.*

If we take a look at delta D_4, we see that it depends on deltas D_2 and D_1. However, since delta D_2 also depends on delta D_1, the dependency between D_4 and D_1 is redundant as it is already implied by the configuration of deltas. Likewise, delta D_5 depends on tree deltas (D_4, D_2 and D_1) of which the dependencies $D_5 \rightarrow D_2$ and $D_5 \rightarrow D_1$ are also implied by the chain of dependencies $D_4 \rightarrow D_2$, and $D_2 \rightarrow D_1$. Therefore, these redundant delta dependencies (indicated by means of a red directed dashed edge in the right graph of deltas) can be safely pruned.

6.5 Conclusion

In this chapter we have described the meta-models and analyses that enable us to represent the history of a system stored in versioning repositories and the changes made to the program entities and relationships present in that history as first-class entities.

By means of a history-based and a change-based representation of the evolution of a system, our approach can later provide a characterization of streams of changes and assist integrators in understanding the changes and their requirements within the stream. Therefore, we can assist the integration of changes across branches and support cherry picking.

We have presented three topics in this chapter. First, we started by presenting other related meta-models that allow the representation of the evolution of a system. We optimized some of their ideas in our history meta-model.

Second, we have described *RingH*, our history meta-model and the analyses that allow us to define and query a history-based representation of the evolution of a system.

Third, we have described *RingC*, our change and dependency meta-model and the delta and dependency analyses needed to define a change-based representation from a history-based representation of a system.

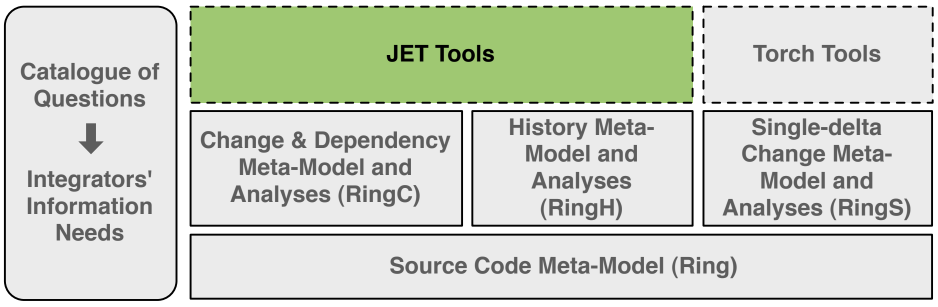
In the next chapter, we present our tool support *JET* built on top of our history and change models. It characterizes a stream of changes in terms of deltas and dependencies, and allows integrators to navigate and query the stream. Moreover, we provide some benchmarks regarding the efficiency and size of *RingC* and the delta and dependency analyses.

CHAPTER 7

JET: Stream Change Analysis in Early Integration Phase

Contents

Contributions Map



Catalogue of Questions
Integrators' Information Needs
JET Tools
Torch Tools
Change & Dependency Meta-Model and Analyses (RingC)
History Meta-Model and Analyses (RingH)
Single-delta Change Meta-Model and Analyses (RingS)
Source Code Meta-Model (Ring)

Overview

This chapter presents *JET*, our approach and (semi-)automated tool support for assisting cross-branch integration. *JET* provides integrators with a characterization of changes and dependencies within a stream of changes, and allows integrators to navigate and query the information required to answer their questions. *JET* is built on top of *RingC* and it is integrated with *Torch*, both described in Section 6.3 and Chapter 5 respectively. First, we explain the characterization of deltas and dependencies. Second, we describe the *JET* tools: the dashboard, the map and the query browser. Third, we discuss how *JET* can aid answering questions regarding how changes from one branch can be merged with another branch. These questions are part of the catalogue presented in Section 3.3.2. Fourth, we present a qualitative assessment of *JET* by means of an evaluation applied to a five-year stream of changes from Squeak with the goal of integrating it into Pharo. The evaluation consisted of two parts: (a) an integrator analyzed and cherry picked changes from the stream, and (b) a developer estimated the effort required for integrating such stream. Fifth, we discuss and compare *JET* with related work.

7.1 Introduction

Assisting integrators in understanding changes across branches and performing cherry picking is very important in a collaborative development environment where the use of branching may require that integrators move changes back and forth between multiple branches. Integrating changes is one of the hardest problems that integrators are faced with, because it usually involves changes from multiple developers and code that the integrator may be unfamiliar with.

In Chapter 5 we have presented *Torch*, our approach and tool support that characterizes changes within a single delta and aims at easing the understanding of such changes. In this chapter we present *JET*, our approach and tool support for integrators that characterizes a stream of changes and aims at assisting integrators at cherry picking changes from the stream. While *Torch* is oriented to aid integrators in understanding changes within a delta, *JET* extends *Torch*'s philosophy to aid integrators at understanding changes and their dependencies within a stream, with the goal of merging across branches.

JET offers a characterization of the changes and dependencies within a stream of changes to provide additional information about these changes that can assist integrators in answering the questions they ask themselves in order to understand and select suitable changes to merge across branches. In Section 3.3.2 we presented a catalogue of questions as a means to identify the integrators' information needs to assist with the integration process. *JET* supports integrators in answering several of these questions by providing integrators the information described in Section 3.4. Such information is accessible by means of simple queries (*e.g.,* changes a certain change relies on, callers of a changed method) complemented by a dedicated dashboard and visualization that aid in comprehending deltas and their dependencies.

JET is a tool built on top of the *RingC* change and dependency meta-model, and our analyses that calculate changes and identify dependencies (as described in Section 6.3 and Section 6.4 respectively). Both analyses take the first-class representation of the a system's history (described in Section 6.2) to generate a first-class representation of the changes and dependencies within that history (*i.e.,* stream), therefore allowing *JET* to characterize the stream of changes.

7.2 Characterizing Deltas and Dependencies within the Stream

JET[1] is our (semi-)automated tool support for assisting cross-branch integration. It allows integrators to contextualize changes, deltas and dependencies within a stream of changes. The goal of this characterization is to speed up the process of understanding changes within a stream, their context and their dependencies, and to support integrators in the decision-making process regarding the integration of changes across branches, especially to assist integrators cherry picking changes. For example, the information provided by *JET* can aid integrators in filtering changes that are irrelevant in a partic-

[1]JET: www.squeaksource.com/JET

ular context and that should not be integrated anyway, in prioritizing which changes to integrate first or last, and so on.

Presence of dependencies. As a first criterion for characterizing dependencies we consider whether a delta has dependencies or not, and the orientation of these dependencies. As mentioned earlier, a delta can depend on other deltas, and a delta can be the dependency of other deltas.

Type of delta	Is dependent	Is a dependency
Source		x
Intermediate	x	x
Island		
End	x	

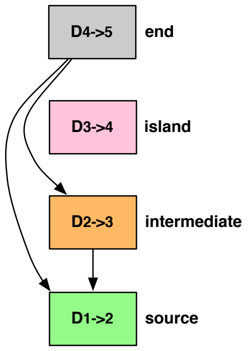


Figure 7.1: *Types of deltas by the presence of dependencies (left) – Example of characterization:* $D_{1\to2}$ *is a* ***source****,* $D_{2\to3}$ *is an* ***intermediate****,* $D_{3\to4}$ *is an* ***island****, and* $D_{4\to5}$ *is an* ***end*** *(right).*

We classify deltas depending on the existence of such dependencies. This classification provides an initial indication of the complexity of a delta and it is therefore potentially valuable to an integrator. In Figure 7.1 we present the four types of deltas (left) along with an illustrative example (right).

- **Island**: a delta that does not depend on another delta and is not the dependency of any delta. *Islands* are the simplest type of delta; integrating them only requires the changes in the delta to be processed.

- **Source**: a delta that has no dependencies but is a dependency of other deltas. *Sources* can still be considered as simple cases as no other changes need to be analyzed beforehand.

- **End**: a delta that depends on other deltas but no other delta depends on it. *Ends* are already complex deltas, because they have to be integrated together with the deltas they depend on.

- **Intermediate**: a delta that depends on, and is the dependency of other deltas. *Intermediates* are the most complex deltas and the ones that should be integrated carefully.

Type and cardinality of change dependencies. The changes belonging to a delta can require the presence of certain source code entities that were introduced (*i.e.,* added entities) or changed (*i.e.,* modified entities) in preceding deltas. As a second criterion, we distinguish between change dependencies that can or cannot be found within the stream.

- **Local**: a dependency is local when the entity it depends on exists within the stream of changes. For example, added class BinaryTree inherits from class Tree, therefore the class BinaryTree depends on class Tree, and class Tree exists within the stream.

- **External**: a dependency is external when the entity it depends on does not exist within the stream. For example, modified method printOn: refers to class Set, but class Set is a library class that was not introduced in the stream of changes.

As we are analyzing object-oriented programming languages, this introduces a level of uncertainty in the case of polymorphic calls (*message sends* in Smalltalk). A polymorphic call can result in the execution of one of several different methods (*i.e.,* implementations of a polymorphic call). The choice is made at run time, and depends on the type of the receiving object (the first argument). As a third criterion, we consider for a particular *change dependency* whether multiple changes that can satisfy such dependency may be present within the stream.

- **Unique**: a change dependency is unique if only one potential change exists in the stream that satisfies the dependency. For example, modified method foo calls method bar, therefore the method foo depends on method bar, and there is only one implementor of bar in the stream.
- **Multiple**: a change dependency is multiple if two or more potential changes exist only in the stream that satisfy the dependency. This is due to polymorphism (*i.e.,* multiple classes implementing the same selector), lack of static type information, and so on. For example, modified method foo calls method bar, and there are four implementors of bar in the stream.

Delta dependency classification. Based on our previous characterization of change dependencies, we also provide a characterization of *delta dependencies*:

- **Needed**: a delta D_1 is a *needed delta dependency* for delta D_2 if at least one change in D_1 is the *unique change dependency* of a change in D_2. In other words, in order to integrate delta D_2, we are certain that we also need to analyze the changes in D_1.
- **Potential**: a delta D_1 is a *potential delta dependency* for delta D_2 if there are changes in D_2 with *multiple change dependencies* and at least one of these change dependencies belongs to D_1. In other words, in order to integrate delta D_2, a developer *will* need to analyze these change dependencies in D_1, to be safe.
- **External**: we say that a delta has *external dependencies* if at least one of its changes requires an entity that is not present within the stream (*e.g.,* a reference to a library).

The dashboard presents deltas and dependencies by using these characterizations. The map, however, only displays needed dependencies to simplify the view. Furthermore, which priority is given to the different types of deltas, or which priority is given to the different types of delta dependencies is up to the developers.

7.3 The JET Tools

Our approach complements the characterization of streams of changes with (semi-)automated tool support – the *JET* tools – that allow integrators to navigate and query a stream of changes *i.e.,* changes, deltas and their dependencies. The *JET* tools offer integrators a means to access their *information needs* in order to answer questions they ask themselves when integrating changes.

The *JET dashboard* presents lists of deltas, lists of changes per delta, lists of dependencies per change, lists of dependencies per delta, and summaries about the number of changes, deltas and dependencies, as can be seen in Figure 7.2. Moreover, the dashboard adds several metrics to each change such as the number of times that an entity was changed, the number of callers and implementors of a method in a single version, or throughout the stream of changes. The *JET query browser* (shown in Figure 7.7) presents the evolution of an entity within the stream, and it complements the information provided by the metrics of each change on the dashboard.

The textual information provided by the dashboard is also complemented by the *JET map*, a visualization displaying deltas with their dependencies, as can be seen in Figure 7.5. The map provides a visual display of a number of metrics such as the number of dependencies of a delta, the number of deltas that depend on a certain delta. By means of a color convention, the map displays a characterization of a set of deltas following the criteria discussed in Section 7.2. Finally, *JET* provides several utilities to developers that allow them to filter and manipulate dependencies and deltas.

In its current state, *JET* is intended to be loaded into the Pharo system in which the integration of changes is performed. This also allows us to access the current working copy (a.k.a image) so that a developer not only can assess a stream of changes with respect to its history, but also with respect to the already integrated source code present in the image.

Philosophy behind the JET tools

- Use the same conventions for the dashboard, query browser and map (when applicable)
- Provide multidirectional navigation of changes, deltas and dependencies on the dashboard
- Always provide access to the source code within the target system on the dashboard and query browser (when applicable)
- Always keep relevant information on the dashboard, query browser and map visible

In the explanation of the *JET* tools we use examples taken from the case study evaluated in Section 7.5 to illustrate several features.

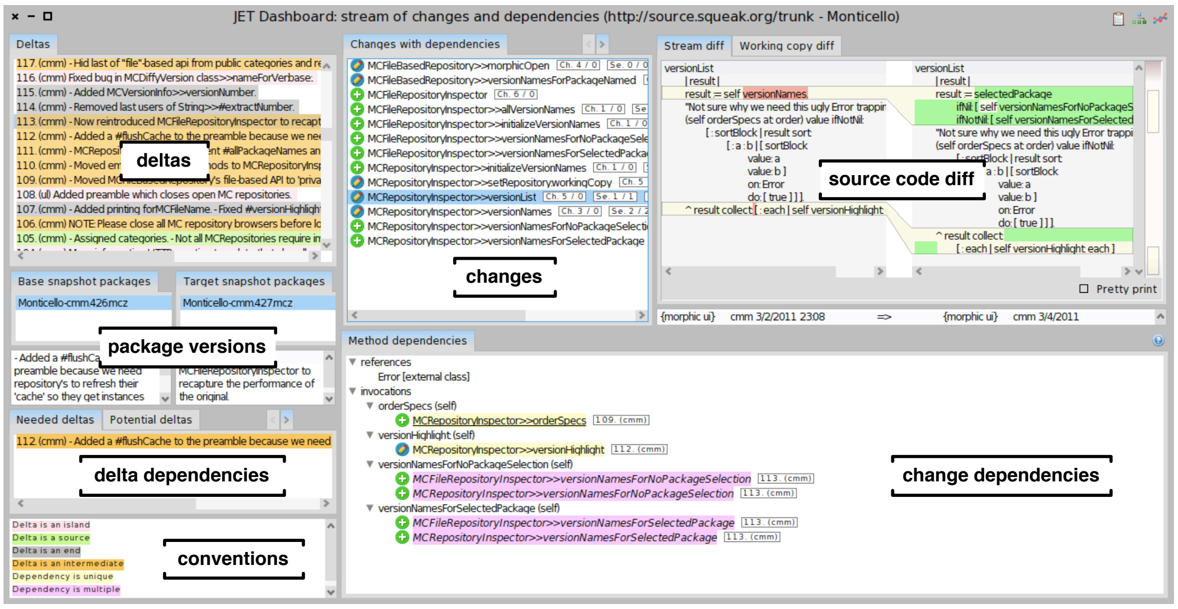
7.3.1 The JET Dashboard

The structure and main elements of the *dashboard* are shown in Figure 7.2. The dashboard offers textual information extracted from the change-based representation of the history of a software system, such as deltas and dependencies, and also allows a developer to access the whole stream in detail.

Deltas. The delta mechanism described in Section 6.4.2 retrieves deltas from a graph of snapshots. These deltas are listed in the first panel of the dashboard (on the top left). Deltas are sorted following the topological order of the snapshots (as described in the import process of the history in Section 6.2.3). To aid integrators in finding the delta they are looking for, the label of each delta is composed of its number[2], the committer, and a summary of the commit message. For example, delta *179.cmm - Fix for package renaming* corresponds to the 179th delta in the stream, committed by *cmm*[3], and it fixed a bug concerning *package renaming*.

[2]The number of a delta is a sequential number assigned to every delta within the stream. For example, if the stream is composed of 100 deltas, the oldest delta is the number 1 and the latest delta is the number 100. This number can be considered as a commit number.

[3]In Pharo, committers often use their initials instead of their full name.



```
JET Dashboard: stream of changes and dependencies (http://source.squeak.org/trunk - Monticello)

Deltas
117. (cmm) - Hid last of "file"-based api from public categories and re
116. (cmm) Fixed bug in MCDiffyVersion class>>nameForVerbase:
115. (cmm) - Added MCVersionInfo>>versionNumber.
114. (cmm) - Removed last users of String>>#extractNumber.
113. (cmm) - Now reintroduced MCFileRepositoryInspector to recapt
112. (cmm) - Added a #flushCache to the preamble because we nee
111. (cmm) - MCReposi                 ent #allPackageNames an
110. (cmm) - Moved em                 hods to MCRepositoryInsp
109. (cmm) - Moved MC                 y's file-based API to 'priva
108. (ul) Added preamble which closes open MC repositories.
107. (cmm) - Added printing forMCFileName. - Fixed #versionHighlight
106. (cmm) NOTE: Please close all MC repository browsers before lo
105. (cmm) - Assigned categories. - Not all MCRepositories require in

Base snapshot packages          Target snapshot packages
Monticello-cmm.426.mcz          Monticello-cmm.427.mcz

- Added a #flushCa              
preamble because we need        MCFileRepositoryInspector to
repository's to refresh their   recapture the performance of
'cache' so they get instances   the original.

Needed deltas | Potential deltas
112. (cmm) - Added a #flushCache to the preamble because we need

Delta is an island
Delta is a source
Delta is an end
Delta is an intermediate
Dependency is unique
Dependency is multiple
```

```
Changes with dependencies
MCFileBasedRepository>>morphicOpen  Ch. 4 / 0  Se. 0 / 0
MCFileBasedRepository>>versionNamesForPackageNamed
MCFileRepositoryInspector  Ch. 6 / 0
MCFileRepositoryInspector>>allVersionNames  Ch. 1 / 0  Se
MCFileRepositoryInspector>>initializeVersionNames  Ch. 1 / 0
MCFileRepositoryInspector>>versionNamesForNoPackageSele
MCFileRepositoryInspector>>versionNamesForSelectedPacka
MCRepositoryInspector>>initializeVersionNames  Ch. 1 / 0
MCRepositoryInspector>>setRepository:workingCopy  Ch. 5
MCRepositoryInspector>>versionList  Ch. 5 / 0  Se. 1 / 1
MCRepositoryInspector>>versionNames  Ch. 3 / 0  Se. 2 / 2
MCRepositoryInspector>>versionNamesForNoPackageSelecti
MCRepositoryInspector>>versionNamesForSelectedPackage
```

```
Stream diff | Working copy diff
versionList                                      versionList
    |result|                                         |result|
    result := self versionNames.                     result := selectedPackage
    "Not sure why we need this ugly Error trappin        ifNil: [ self versionNamesForNoPackageS
    (self orderSpecs at: order) value ifNotNil:          ifNotNil: [ self versionNamesForSelected
        [ :sortBlock | result sort:                  "Not sure why we need this ugly Error trappi
            [ :a :b | [ sortBlock                    (self orderSpecs at: order) value ifNotNil:
                value: a                                 [ :sortBlock | result sort:
                value: b ]                                   [ :a :b | [ sortBlock
                on: Error                                        value: a
                do: [ true ] ] ].                                value: b ]
    ^ result collect: [ :each | self versionHighlight            on: Error
                                                                 do: [ true ] ] ].
                                                     ^ result collect:
                                                         [ :each | self versionHighlight: each ]
☐ Pretty print
{morphic ui}   cmm 3/2/2011 23:08   =>   {morphic ui}   cmm 3/4/2011
```

```
Method dependencies
▼ references
      Error [external class]
▼ invocations
    ▼ orderSpecs (self)
          MCRepositoryInspector>>orderSpecs  109. (cmm)
    ▼ versionHighlight: (self)
          MCRepositoryInspector>>versionHighlight:  112. (cmm)
    ▼ versionNamesForNoPackageSelection (self)
          MCFileRepositoryInspector>>versionNamesForNoPackageSelection  113. (cmm)
          MCRepositoryInspector>>versionNamesForNoPackageSelection  113. (cmm)
    ▼ versionNamesForSelectedPackage (self)
          MCFileRepositoryInspector>>versionNamesForSelectedPackage  113. (cmm)
          MCRepositoryInspector>>versionNamesForSelectedPackage  113. (cmm)
```

Figure 7.2: *The JET dashboard and its main elements.*

Package versions. Each delta represents the changes that happened between a *base* and a *target* snapshot (*i.e.,* predecessor and successor within the graph of snapshots). This panel lists separately the package versions included in each snapshots, and for each package version the complete commit message is presented as well. Monticello uses the following convention to name package versions: *package name-commiter.revision number*. For example, *Compiler-MarkusDenker.293* corresponds to the *293rd* revision of the package *Compiler* committed by *MarkusDenker*.

Changes. The changes to packages, classes and methods belonging to a delta are classified into two lists: *Changes with dependencies* and *Changes without dependencies*. Both lists allow a developer to inspect all changes of a delta and their evolution within the stream.

Each change in the list is accompanied by metrics such as the number of times that the entity changed or the number of changes that appear in later deltas with regard to the delta under analysis (*i.e.,* changes to the entity that happened later in the history). This information aims at speeding up the analysis of changes that affected the same entity by indicating for example that a particular change is not the latest one within the stream. An integrator can then inspect the whole evolution of such an entity instead of trying to understand each change separately.

A change to any kind of entity has the metric *number of changes* (`Ch`) corresponding to the number of times that the entity changed, *e.g.,* `Ch 5/2` corresponds to 5 changes over the total stream and 2 more changes in two later deltas.

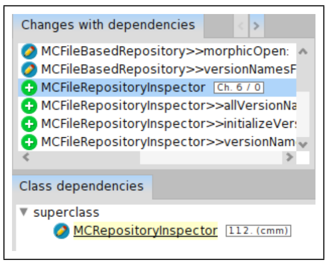
A change to a method `m` has two other metrics that take into account information from the target snapshot from which originated the delta under analysis. By means of this, we query the complete view of the system where the changed was applied. We refer to a delta's target snapshot as delta's snapshot. The first metric is *number of callers*[4] (`Se`), for example: `Se 0/1` corresponds to 0 methods calling `m` in the delta's snapshot, and 1 method calling `m` in a later delta's snapshot. The second metric

[4] In Smalltalk, the callers of a method are known as senders.

is *number of implementors* (`Im`), for example: `Im 3/5/4` corresponds to 3 classes implementing a method with selector[5] m in the delta's snapshot, 5 classes implementing a selector m in a later delta's snapshot, and 4 classes implementing a selector m in the working copy[6].

Source code diff. The source code of a change is shown in a panel named *Stream code* or *Stream diff*. The first appears for *additions* and *removals* and shows the plain source code that was added or removed. The second appears for *modifications* and shows a *diff* highlighting the part of the code that changed (in red or green for added and removed respectively). Moreover, if the changed entity (*e.g.,* method) exists in the working copy, another diff (*Working copy diff*) will appear comparing both. By providing the *Working copy diff* an integrator not only can inspect the code that changed within the stream but can also compare that code to the current code of the system. Finally, additional information about the change is displayed, such as the author that changed the entity and the timestamp of the change. Other information will appear depending of the kind of entity, *e.g.,* the protocol of a method.

Change dependencies. This panel shows the change dependencies of methods and classes grouped by invocations (method calls), class references and superclasses (as shown in Figure 7.2). Each change dependency indicates the change associated to it and the delta to which that change belongs.


Changes with dependencies
MCFileBasedRepository>>morphicOpen:
MCFileBasedRepository>>versionNamesF
MCFileRepositoryInspector Ch. 6 / 0
MCFileRepositoryInspector>>allVersionNa
MCFileRepositoryInspector>>initializeVer:
MCFileRepositoryInspector>>versionNam
Class dependencies
superclass
MCRepositoryInspector 112. (cmm)

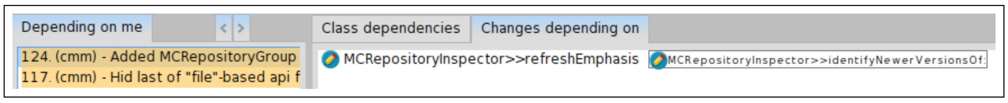

Depending on me
124. (cmm) - Added MCRepositoryGroup
117. (cmm) - Hid last of "file"-based api f
Class dependencies
Changes depending on
MCRepositoryInspector>>refreshEmphasis
MCRepositoryInspector>>identifyNewerVersionsOf:

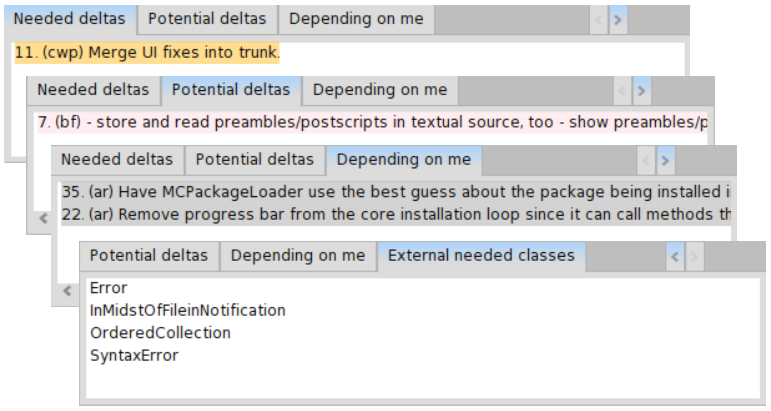
Figure 7.3: *Example: exploring the change dependencies of an **added** class within delta* `113.cmm` *(top) – Exploring why delta* `124.cmm` *depends on delta* `113.cmm` *(bottom).*

An example is illustrated in Figure 7.3. On the top it shows that the *added* class MCFileRepositoryInspector of delta `113.cmm` depends on the superclass MCRepositoryInspector (*i.e.,* its superclass). Since this superclass was most recently modified in delta `112.cmm`, there exists a change dependency between *added* MCFileRepositoryInspector and *modified* MCRepositoryInspector, and it implies the delta dependency `113.cmm` → `112.cmm`.

From this panel, an integrator can also filter change dependencies, or can inspect which of the changes of one delta are a dependency of another delta, *e.g.,* Figure 7.3 (on the bottom) shows that delta `124.cmm` depends on delta `113.cmm` (left), and that the *modified* method MCRepositoryInspector»refreshEmphasis[7] of `124.cmm` calls the *modified* method MCRepositoryInspector»identifyNewerVersionsOf: of `113.cmm` (right) causing the delta dependency `124.cmm` → `113.cmm`.

[5]The name of a method is known as *selector* in Smalltalk.
[6]The working copy is the code loaded in a Pharo image.
[7]In Pharo, the usual convention to refer to methods is ClassName»methodName.

Delta dependencies. This panel presents four lists of dependencies for a particular delta. Three correspond to the characterization of delta dependencies based on the categories discussed in Section 7.2: *Needed dependencies*, *Potential dependencies*, and *External dependencies*. To ease navigation of every dependency related to a delta, the fourth list shows the *Deltas depending on me* which include the deltas that depend on a particular delta. Figure 7.4 illustrates the delta dependencies of delta `12.ar`: (a) it needs the *intermediate* delta `11.cwp`, (b) it potentially needs the *island* delta `7.bf`, (c) *end* deltas `22.ar` and `35.ar` depend on it, and (d) it has four external dependencies to classes Error, InMidstOfFileinNotification, OrderedCollection and SyntaxError.



11. (cwp) Merge UI fixes into trunk.
7. (bf) - store and read preambles/postscripts in textual source, too - show preambles/p
35. (ar) Have MCPackageLoader use the best guess about the package being installed i
22. (ar) Remove progress bar from the core installation loop since it can call methods th
Error
InMidstOfFileinNotification
OrderedCollection
SyntaxError

Figure 7.4: *Example: delta dependencies of delta* `12.ar`*.*

Conventions. Colors are used to represent the types of deltas and dependencies described in Section 7.2. They help developers get instantaneous information and reinforcement of their knowledge. The conventions are the same in the entire dashboard: pink for *island* deltas, green for *source* deltas, grey for *end* deltas, orange for *intermediate* deltas, yellow for *unique* change dependencies and magenta for *multiple* change dependencies. Font styles are used to complement dependencies, italic for change dependencies within the same delta, and underlining for *redundant* dependencies. Icons are also used to represent each kind of change: green plus for additions, blue pencil for modifications and red minus for removals. These icons are the same used in *Torch* to represent additions, modifications and removals.

Integration with Torch

In Chapter 5 we described *Torch*, our tool support that allows integrators to understand changes and their context within a single delta. *JET* aims at providing the same aid but it is augmented for a stream of changes (*i.e.,* set of deltas). The main goal is to assist the integrating of changes across branches, and therefore assist cherry picking.

JET integrates *Torch* with the dashboard in order to provide integrators with extended support. Integrators can first have an overview of the changes and dependencies within a stream of changes by means of *JET*, and then have an in-depth understanding of the changes of a particular delta by means of *Torch*. Note that *Torch* does not need to be used for understanding each single delta, as *JET* already provides valuable information about changes. However, depending on the complexity of a particular delta *Torch* can play an important role to ease its understanding, for example in cases when a delta consists of a large number of changes, displaying the context of such changes provides a better view of the changes.

Considering that *Torch* makes use of the *RingS* meta-model described in Section 5.5 to represent the changes within a single delta, *JET* provides *Torch* with the base and target snapshots that generate the delta instead of the delta itself.

7.3.2 The JET Map

The *map* is a visualization that aims at providing an overview of deltas and their dependencies, and guiding integrators in determining where to start the analysis of a stream of changes. The map, as shown in Figure 7.5, mainly offers a simplified view of the dashboard information in order to give an initial insight of the requirements of deltas. In a sense, this visualization provides integrators with a means to assess how complex it is to integrate the changes of a particular delta.

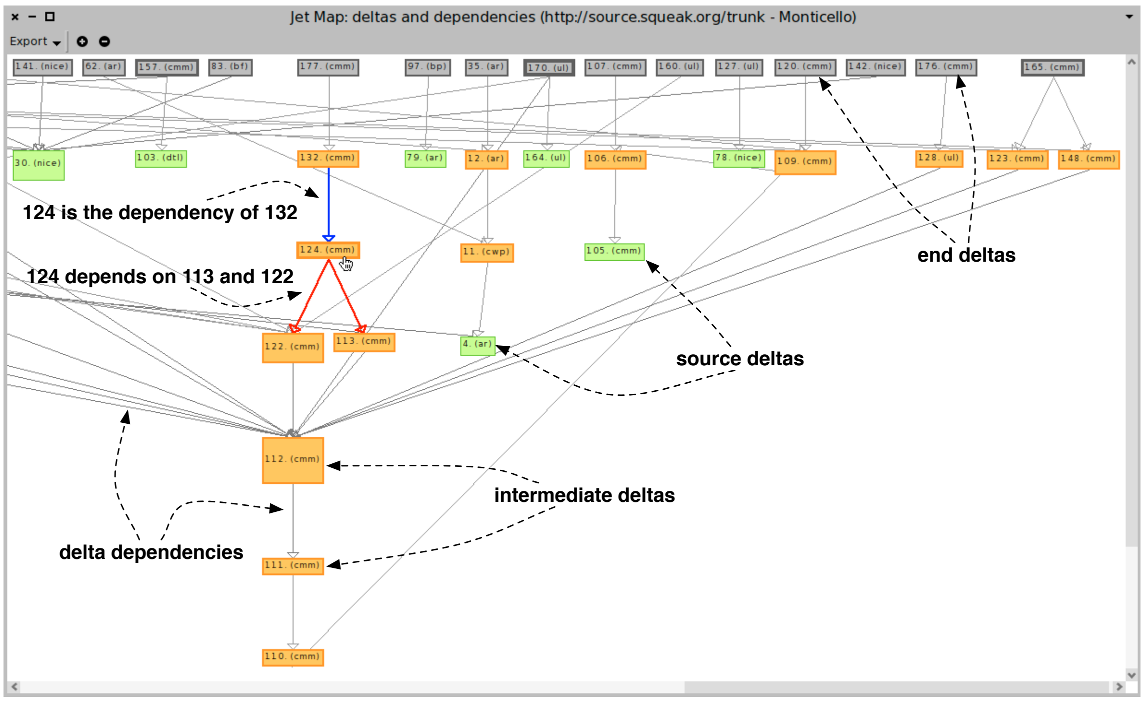


Figure 7.5: *The JET map: green nodes are **source** deltas (not depending on others), orange nodes are **intermediate** deltas (having dependencies and others depending on them) and grey nodes are **end** deltas (only depending on others).*

By means of the map, we provide information about a stream of changes that can support integration across branches. Among the integrator's information needs described in Section 3.4, we identified *change dependencies* as part of the *historical information* required to answer several questions. The map shows dependencies between deltas to allow integrators to quickly identify which dependencies are required by a particular set of changes (*i.e.,* commit).

The map only visualizes deltas that have dependencies or serve as dependencies of other deltas. That is, it shows *source*, *intermediate* and *end* deltas, and omits *island* deltas. Rectangles are used to represent deltas and directed edges to represent dependencies. A rectangle includes the label of a delta (number and committer). The height of a rectangle (delta) is related to the number of deltas that depend on this delta. The border width of a rectangle is related to the number of dependencies of this delta. The map uses the same color conventions as the dashboard (*i.e., green* for sources, *grey* for ends and *orange* for intermediates).

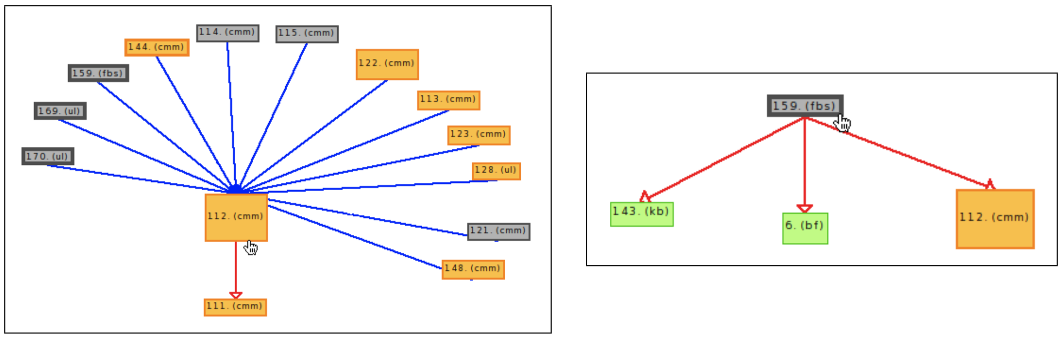


Figure 7.6: *Deltas and dependencies on the map.*

The map also allows integrators to navigate over the deltas. It uses *red directed edges* to point to the dependencies of a delta, and *blue directed edges* to indicate which deltas depend on a particular delta. Figure 7.6 takes a more in-depth view of two deltas on the map. The example on the left displays the *intermediate* delta `112.cmm` (the orange node in the middle) that only depends on the *intermediate* delta `111.cmm`, therefore the border of `112.cmm` is thin. A red directed edge indicates this dependency. On the top, we see twelve deltas that depend on `112.cmm` which makes the rectangle considerably taller than the other visualized deltas. Blue edges are highlighted to indicate these dependencies. The example on the right displays the *end* delta `159.fbs` (the grey node at the top) that has three dependencies on *source* deltas `143.kb` and `6.bf` (green nodes), and on *intermediate* delta `112.cmm` (orange node). Thus the border of the node in this case is thicker compared to the previous example. As this is an *end* delta, meaning that no deltas depend on it, the height of the rectangle has the smallest possible value.

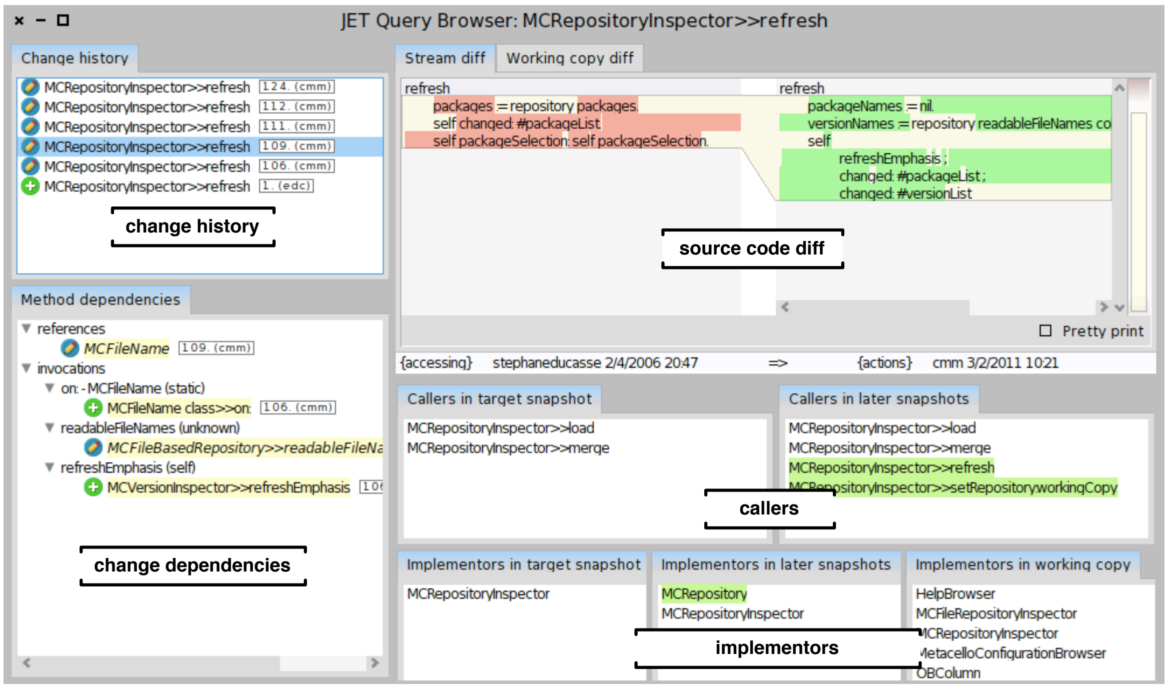
Finally, the map also offers textual information as a *fly-by-help* when the integrators navigate over the deltas and dependencies. For a delta it shows the commit messages and for a dependency it shows the deltas involved and their commit messages. Note that this is not shown in Figure 7.6.

7.3.3 The JET Query Browser

JET complements the information provided by the dashboard and the map with a third browser, the *query browser* that provides more fine-grained information about the changes. This browser aims at aiding integrators in understanding the complete evolution of an entity, together with its dependencies and users at any point in time (*i.e.,* a delta) within the stream.

The *historical information* described in Section 3.4 is key information to accomplish our characterization of streams of changes. The query browser provides integrators with this information which is needed to answer many questions presented in the catalogue in Section 3.3.2. They are related to changes within a stream, such as "Is this change still the most recent one?" or "Is there any later change in the sequence that supersedes it?".

The structure of the query browser is shown in Figure 7.7. Note that it follows the same conventions (color, font styles and icons) and reuses two components of the dashboard (*change dependencies* and *source code diff*), both described in the Section 7.3.1. In the following we describe each of the components of the query browser.


JET Query Browser: MCRepositoryInspector>>refresh
Change history
Stream diff
Working copy diff
Method dependencies
Pretty print
Callers in target snapshot
Callers in later snapshots
Implementors in target snapshot
Implementors in later snapshots
Implementors in working copy
change history
source code diff
change dependencies
callers
implementors

```
MCRepositoryInspector>>refresh 124. (cmm)
MCRepositoryInspector>>refresh 112. (cmm)
MCRepositoryInspector>>refresh 111. (cmm)
MCRepositoryInspector>>refresh 109. (cmm)
MCRepositoryInspector>>refresh 106. (cmm)
MCRepositoryInspector>>refresh 1. (edc)
```

```
▼ references
      MCFileName 109. (cmm)
▼ invocations
   ▼ on: - MCFileName (static)
         MCFileName class>>on: 106. (cmm)
   ▼ readableFileNames (unknown)
         MCFileBasedRepository>>readableFileNa
   ▼ refreshEmphasis (self)
         MCVersionInspector>>refreshEmphasis 10
```

```
refresh                                          refresh
    packages := repository packages.                 packageNames := nil.
    self changed: #packageList.                      versionNames := repository readableFileNames co
    self packageSelection: self packageSelection.    self
                                                         refreshEmphasis ;
                                                         changed: #packageList ;
                                                         changed: #versionList

{accessing}   stephaneducasse 2/4/2006 20:47   =>   {actions}   cmm 3/2/2011 10:21
```

```
MCRepositoryInspector>>load
MCRepositoryInspector>>merge
```

```
MCRepositoryInspector>>load
MCRepositoryInspector>>merge
MCRepositoryInspector>>refresh
MCRepositoryInspector>>setRepositoryworkingCopy
```

```
MCRepositoryInspector
```

```
MCRepository
MCRepositoryInspector
```

```
HelpBrowser
MCFileRepositoryInspector
MCRepositoryInspector
MetacelloConfigurationBrowser
OBColumn
```

Figure 7.7: *The JET query browser and its elements.*

Change history. The changes of an entity are listed in the first panel of the browser. This shows how an entity has evolved within the stream, and therefore already answers several questions related to frequency of change, who are the authors, in which version the entity change, when was the entity changed, which are the later changes of that entity. For each change, the *kind of action* is shown by means of an icon (addition, modification, removal) as well as the delta in which the change occurred. Note that this component provides the currently selected context of the browser as seen in Figure 7.7.

Source code diff. This component is the same as the *source code diff* described in Section 7.3.1. The advantage of this component in the context of the query browser is that for a particular entity the integrator is able to explore how the code evolved, together with who (author) changed the code and when (timestamp). That means, within one browser the whole sequence of changes applied to the same entity can be textually compared.

Change Dependencies. The component lists dependencies of methods or classes. It is the same *change dependencies* component of the dashboard explained in Section 7.3.1. While in the dashboard changes are grouped per delta, the query browser list the sequence of changes made to the same entity, and therefore allows integrators to compare how the dependencies of a particular change evolved.

Callers. This panel is shown when exploring the evolution of a method. For a method m, the callers correspond to the set of methods that are invoking method m. Callers are found in the target snapshot that caused the delta (current delta) to which the change under analysis belongs. By means of this, we query the callers in the complete view of the system where the change was applied. In what follows, when we mention the delta's snapshot, we mean the target snapshot of the delta

Two lists are used to present the callers of a method. The first list shows the callers that exist in the current delta's snapshot. The second list shows the callers that exist in subsequent delta's snapshots (later points in the stream). This shows whether a method is actually used within the delta and also how important it is for the subsequent deltas. Callers that are removed later in the stream appear with a light red background in the first list. Callers that are added later in stream appear with a light green background in the second list.

Implementors. This panel is also shown when exploring the evolution of a method. For a method m, the implementors correspond to the set of classes that define a method with selector m. We extract the implementors following the same logic as for callers, *i.e.,* we use the current delta and the delta's snapshot.

Implementors are presented in three lists. As for callers, the first two lists present the implementors found in the current delta's snapshot, and in subsequent delta's snapshots (later points in the stream), respectively. Both lists follow the same color convention as well. The third list complements the historical information by showing the implementors existing in the working copy. An integrator can compare the classes that implement a selector within the stream of changes to the current classes of the system (in Pharo) that implement the same selector.

7.3.4 How to Use the JET Tools

Here we describe the intended process of using *JET* to assist the integration of changes.

1. To make use of *JET* an integrator is required to load the *JET* tools[8] into a Pharo image containing the latest version of the Pharo system where the integration is done. This corresponds to the current system (*i.e., target branch*) of the integration process.

2. The complete or partial history of the feature or system that will be integrated into Pharo needs to be imported (described in Section 6.2.3). This corresponds to creating the history-based representation of the *stream of changes* from the *source branch*.

3. The delta and dependency analyses described in Section 6.4 need to be performed to create a change-based representation (*i.e.,* deltas an dependencies) of the history resulting in the previous step.

4. Provide to the dashboard the deltas and dependencies representing the stream of changes for characterizing the stream.

The dashboard provides the characterization of deltas and dependencies, and it is the entry point to the map and the query browser. The toolbar at the top of the dashboard has icons to access the map and the history browser. This simple auxiliary tool allows integrators to explore the history-based representation of the stream in detail. Each of the changes listed on the dashboard allows an integrator to access the query browser, and each of the deltas allows access to the *Torch* dashboard where the changes within that delta are characterized and visualized. Other utilities to filter and manipulate dependencies and deltas are available in the dashboard.

Because the *JET* tools are under evaluation, they are not currently integrated with other tools within the Pharo environment. For example, with the version control system. The analysis of a

[8]The JET tools: www.squeaksource.com/JET

stream of changes is performed in isolation. From this analysis, integrators can decide which changes are suitable for integration, and later using the version control system to perform the actual merge. Note that the integrator is free to use the version control system and browsing facilities provided by Pharo to complement his analysis regarding the potential impact of such changes on the target system.

Providing support to perform the actual cherry picking and merging within the *JET* tools, and therefore integrating *JET* with the version control system is an avenue for future work.

7.4 Supporting the Answering of Integrator Questions

In this Section, we discuss how *JET* can aid in answering the questions that integrators ask themselves when performing integration activities. We introduced a catalogue of 64 questions in Section 3.3 that served to identify the integrators' information needs in order to assist them during the integration process. These questions also serve as motivation for several of the features provided by the dashboard and query browser such as the list of changes, metrics per change, list of dependencies per change or delta, source code diffs, etc. These features provide integrators with the needed information and ease the answering of their questions.

We guide our discussion based on the 5 categories used to classify the questions: (a) author/owner questions, (b) behavioral questions, (c) structural change characterization questions, (d) infrastructure questions, and (e) temporal and change stream questions.

Author questions. Answering five of the six questions in this group can easily be done with *JET* (*e.g.,* "Who wrote the original code that was changed" or "Who made this change?"). The author of any change and the committer of a group of changes are shown in the dashboard and query browser. The committers are also shown on the map. Moreover, the query browser (described in Section 7.3.3) allows integrators to identify how many developers and who have changed an entity within the stream. For example, the query browser shown in Figure 7.7 presents the evolution of the method MCRepositoryInspector»refresh. It was changed by two developers: `edc` who introduced this method in delta `1.edc`, and `cmm` who modified it later in deltas `106.cmm`, `109.cmm`, `111.cmm`, `112.cmm` and `124.cmm`. Answering "What is the general quality of the change committer?" is subjective and it is up to the integrator to establish the quality of a committer.

Behavioral questions. Most of the questions in this category are not supported by *JET*. Only 6 out of the 14 questions in this group can be partially supported. For example, for answering questions such as "What is the reason of this change?" or "What are the implications of this change for API clients?" *JET* does not provide straightforward answers. The integrator can use the information provided by the tools but it is up to him to find the answers. However answering "What kind of change is it? (Bugfix/New feature/Refactoring/Documentation)" depends on the complexity of the change and on how the committer coupled the modifications that collaborate to the same change instead of committing multiple unrelated changes. Moreover a single kind of change may be represented by a sequence of commits making the identification of the change troublesome. New features or removals could be easily identified as the tools show when entities are added, modified or removed, along with their structure *e.g.,* knowing that a delta contains mostly additions of code are likely the introduction of a new feature. The access to *Torch* from within *JET* to explore a single delta also eases in answering this question. The other 8 questions are related to test coverage which is not part of our approach and therefore answering “Is the change covered by tests? What is the coverage?”, “Did this change fix/break tests? Which tests?” or “Did the tests work before the changes?” is not possible.

Structural change characterization questions. Most of the simple questions in this category are directly covered by *JET*. 14 questions can be fully answered using the dashboard and taking advantage of its integration with the *Torch* dashboard (described in Chapter 5). These questions are related to change size (“How large is the change?”), change scope and structure (“What is the scope of this change (which/how many classes/packages/..., local/global?” or “Is this change confined to a single package?”), change dependencies (“What are the required structural dependencies?”), and correlation of changes (“Are there other packages that would need to change as well to incorporate this change?”).

Six questions are more challenging and *JET* does not offer complete support for answering them. For example, support for answering the question “What is the complexity of the changes / of the touched classes?” could be improved. To give the integrator an initial idea of the complexity of a change, we could for example use metrics such as *Cyclomatic complexity*. Note however that this does not necessarily provide a good approximation of the complexity of change as this metric is structurally-oriented. Some changes with a low cyclomatic complexity can still pose a real challenge to integrate because of a complex interaction with the system.

The question “Can I apply this change?” is also hard to answer and is partially supported. *JET* uses the change and dependency model to identify mostly structural dependencies in order to find which changes are required to apply/integrate a particular change. However, *JET* does not perform behavioral analysis in the sense of Reuse Contracts [Steyaert 1996] for example. In addition, since Smalltalk is dynamically-typed, static analysis on the integrated changes cannot be performed. Note that, if we were supporting statically-typed languages such as Java, it would be possible to perform static analysis. We plan to investigate this in future work.

The implication of the question “What parts of the system are directly using the changed behavior?” is challenging. Indeed an integrator wants to assess the impact of a change on the system. *JET* provides some support for that based on a model that does not take types into account. Therefore, lots of false positives may be reported in the presence of heavy polymorphic invocations that need to be manually processed by the integrator.

The question “Does the change follow rule checking/conventions?” is not supported by *JET* at the level of rule checking. However, rule checker results could easily be integrated with *JET*. Finally, the question “Is the vocabulary used in the change consistent with the one of the system?” can also be partially supported by analyzing the vocabulary introduced with the change. Note that *JET* provides such information by means of the symbolic clouds offered by *Torch* (described in Section 5.3.5). This however could be improved to provide an assessment of the vocabulary of changes regarding the system.

Infrastructure questions. Two questions regarding the bug tracking infrastructure such as “Have other bugs related to the change been reported?” are not supported by *JET*. In its current incarnation, our approach only analyzes the source code of a system, and *JET* provides characterization of changes and dependencies extracted from the source code. Therefore questions involving other artifacts like documentation or bug reports are not taken into account. They lie outside the scope of our approach.

Temporal and change stream questions. Most of the questions in this category are covered by *JET*. 22 out of 23 questions can be fully answered. The package versions feature of the dashboard and the source code diffs show the commit timestamps and the timestamps at which any change occurred. Therefore answering questions such as "Did this method/feature change (a lot) recently/in the past?" or "Did this change ever happen before?" is trivial.

The dashboard and query browser are dedicated to answering these questions. Especially, the query browser that shows the history of an entity and the change metrics play a fundamental role in answering temporal questions. For example, the question "Is the change ever used in subsequent changes?" can immediately be answered by an user of our tool using the metrics (*number of callers*) shown with each change. Similarly, answering "Is this change part of a whole series of changes?" or "Does this change depend on previous ones?" is explicitly supported by the characterization of deltas based on dependencies.

The question "Was this method/class renamed in the past? in which version?" is a challenging one as no renamings are directly identified and modeled with our *RingC* change meta-model, and therefore *JET* does not provide such information. However, this can be improved considering the fine-grained information represented by our *RingH* history meta-model (both models described in Chapter 6).

7.5 Qualitative Evaluation: Integrating Monticello Changes into Pharo

In this section we present a qualitative assessment of our approach, in particular we evaluate how the characterization of a stream of changes and tool support assist cross-branch integration. As a case study, we considered the integration of the latest changes of the Squeak branch of Monticello into the Pharo system. Our case study consists of two parts. In the first part, we asked one of the Pharo *integrators* to use the *JET* tools while integrating (parts of) Monticello. For the second part, we asked a *developer* knowledgable in Monticello to use *JET* to *estimate the effort* required for integrating the Squeak branch of Monticello with Pharo. While the former part of the case study provides us some insights into the perceived usefulness of the different features of *JET*, the latter part aims at assessing the effort and time required to use *JET* for analyzing a part of the history of Monticello in Squeak.

Note that this case study does not allow us to make any generalizable claims regarding the usefulness of *JET*. Given the challenges associated with change integration, a full-fledged validation would require a controlled experiment with *advanced* developers (instead of for example groups of master students). We consider such an experiment as an avenue for future work and it will be discussed in Section 8.4.

7.5.1 Case Study Description: Monticello Version Control System

Figure 7.8 shows the context of our case study, concretely two streams of changes of the Monticello version control system. After forking Pharo from Squeak in 2008, Pharo developers modified their own branch of the Monticello core package[9] (268 commits), while the Squeak developers continued

[9]The Monticello core package excludes UI and tests.

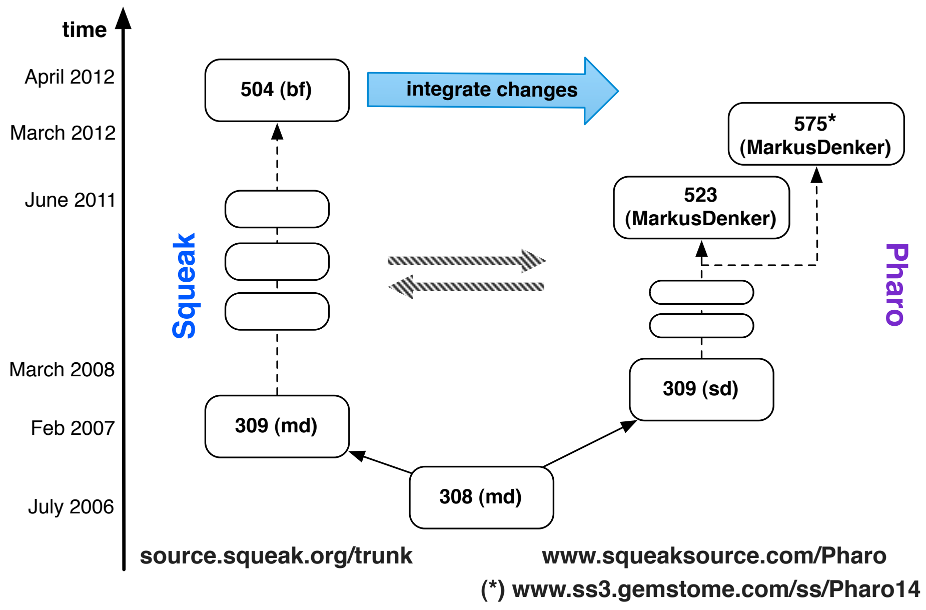


Figure 7.8: *Monticello - two streams of parallel changes of the core package (on April 23rd, 2012).*

the development of the core package (196 commits) in the original repository. Although some of the changes in the Squeak branch were already integrated into Pharo, this process occurred in an entirely ad-hoc manner.

The Monticello core package implements the version control system used by Squeak and Pharo: streaming in/out of code, code representation, ancestor analysis, textual diff tools, three-way merge, working copy diffing, remote distributed code repositories and their management. This package is complemented by a UI (Monticello Tools[10]) and tests. Table 7.1 shows the size of the Monticello version control system (core package) in Squeak and Pharo.

Description	In Squeak	In Pharo
Classes	117	116
Methods	1559	1587
Lines of code	7739	8083

Table 7.1: *Size of the Monticello core package (on April 23rd, 2012).*

Prior to performing the case study, we loaded the history of the Squeak branch of Monticello (from February 2007 to April 2012) into *JET*[11]. Table 7.2 gives an overview of several metrics provided by *JET* regarding changes, deltas and dependencies, along with the memory footprint and creation time. Note that the total deltas shown on the table is 193 instead of 196 because three versions of the Monticello core package were missing from the Squeak repository. The last two rows show the memory footprint of the calculation of deltas and change dependencies and the time this took[12]. While we do not claim that our approach scales, these numbers seem to indicate that the computational overhead of our approach is limited.

[10]The Monticello Tools: https://gforge.inria.fr/frs/download.php/27018/Monticello.pdf

[11]An image containing *JET* and the case study can be found at: http://soft.vub.ac.be/~vuquilla/JET-Pharo-1.3-13328-OneClick.zip

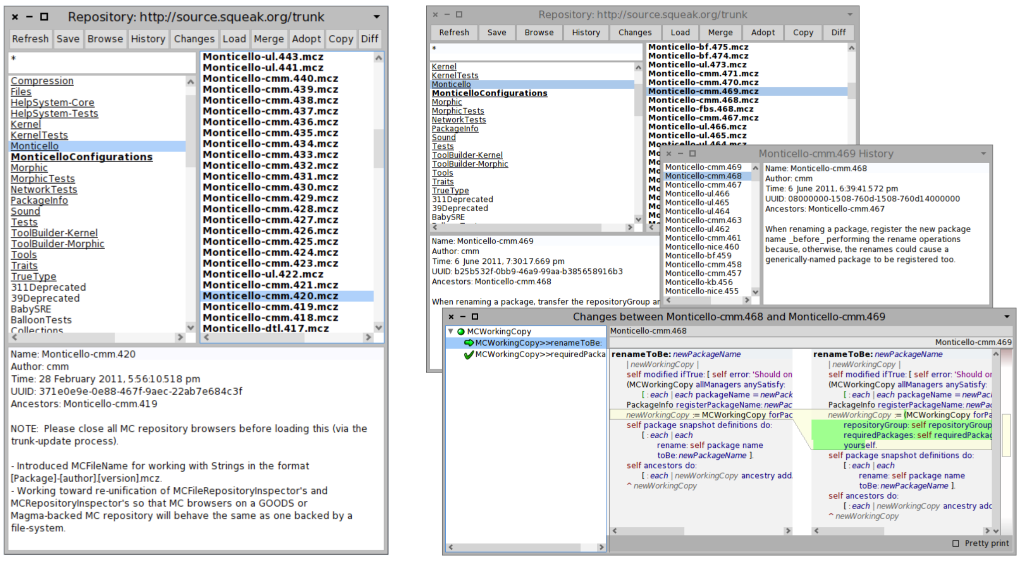
[12]On an Apple MacBook Pro with an Intel Core 2 Duo 2.8GHz processor and 4GB of RAM.

Description	#
Additions	2354
Modifications	593
Removals	601
Total changes	*3548*
Changes *with* dependencies	1909
Changes *without* dependencies	1639
Changes *with external* dependencies	607
Change dependencies	11530
Delta dependencies	111
External dependencies	122
Intermediate deltas (orange)	18
Island deltas (pink)	105
End deltas (grey)	48
Source deltas (green)	22
Total deltas	*193*
Memory footprint	2.46 Mb
Object creation time	31298 ms

Table 7.2: *Metrics: changes, deltas, dependencies, memory and time.*

7.5.2 Part 1: Integrator Experiences

As mentioned earlier, the first part of our case study consisted of observing an experienced Pharo integrator – Stéphane Ducasse – using *JET* while integrating changes from the Squeak branch of Monticello into Pharo.



Repository: http://source.squeak.org/trunk
Refresh
Save
Browse
History
Changes
Load
Merge
Adopt
Copy
Diff
Monticello-cmm.469 History
Changes between Monticello-cmm.468 and Monticello-cmm.469
Pretty print

```
*
Compression
Files
HelpSystem-Core
HelpSystem-Tests
Kernel
KernelTests
Monticello
MonticelloConfigurations
Morphic
MorphicTests
NetworkTests
PackageInfo
Sound
Tests
ToolBuilder-Kernel
ToolBuilder-Morphic
Tools
Traits
TrueType
311Deprecated
39Deprecated
BabySRE
BalloonTests
Collections
```

```
Monticello-ul.443.mcz
Monticello-ul.441.mcz
Monticello-cmm.440.mcz
Monticello-cmm.439.mcz
Monticello-cmm.438.mcz
Monticello-cmm.437.mcz
Monticello-cmm.436.mcz
Monticello-cmm.435.mcz
Monticello-cmm.434.mcz
Monticello-cmm.433.mcz
Monticello-cmm.432.mcz
Monticello-cmm.431.mcz
Monticello-cmm.430.mcz
Monticello-cmm.429.mcz
Monticello-cmm.428.mcz
Monticello-cmm.427.mcz
Monticello-cmm.426.mcz
Monticello-cmm.425.mcz
Monticello-cmm.424.mcz
Monticello-cmm.423.mcz
Monticello-ul.422.mcz
Monticello-cmm.421.mcz
Monticello-cmm.420.mcz
Monticello-cmm.419.mcz
Monticello-cmm.418.mcz
Monticello-dtl.417.mcz
```

```
Name: Monticello-cmm.420
Author: cmm
Time: 28 February 2011, 5:56:10.518 pm
UUID: 371e0e9e-0e88-467f-9aec-22ab7e684c3f
Ancestors: Monticello-cmm.419

NOTE:  Please close all MC repository browsers before loading this (via the
trunk-update process).

- Introduced MCFileName for working with Strings in the format
[Package]-[author].[version].mcz.
- Working toward re-unification of MCFileRepositoryInspector's and
MCRepositoryInspector's so that MC browsers on a GOODS or
Magma-backed MC repository will behave the same as one backed by a
file-system.
```

```
*
Kernel
KernelTests
Monticello
MonticelloConfigurations
Morphic
MorphicTests
NetworkTests
PackageInfo
Sound
Tests
ToolBuilder-Kernel
ToolBuilder-Morphic
Tools
Traits
TrueType
311Deprecated
39Deprecated
BabySRE
```

```
Monticello-bf.475.mcz
Monticello-bf.474.mcz
Monticello-ul.473.mcz
Monticello-cmm.471.mcz
Monticello-cmm.470.mcz
Monticello-cmm.469.mcz
Monticello-cmm.468.mcz
Monticello-fbs.468.mcz
Monticello-cmm.467.mcz
Monticello-ul.466.mcz
Monticello-ul.465.mcz
```

```
Name: Monticello-cmm.469
Author: cmm
Time: 6 June 2011, 7:30:17.669 pm
UUID: b25b532f-0bb9-46a9-99aa-b385658916b3
Ancestors: Monticello-cmm.468

When renaming a package, transfer the repositoryGroup ar
```

```
Monticello-cmm.469
Monticello-cmm.468
Monticello-cmm.467
Monticello-ul.466
Monticello-ul.465
Monticello-ul.464
Monticello-cmm.463
Monticello-ul.462
Monticello-cmm.461
Monticello-nice.460
Monticello-bf.459
Monticello-cmm.458
Monticello-cmm.457
Monticello-kb.456
Monticello-nice.455
```

```
Name: Monticello-cmm.468
Author: cmm
Time: 6 June 2011, 6:39:41.572 pm
UUID: 08000000-1508-760d-1508-760d14000000
Ancestors: Monticello-cmm.467

When renaming a package, register the new package
name _before_ performing the rename operations
because, otherwise, the renames could cause a
generically-named package to be registered too.
```

```
▼ MCWorkingCopy
    MCWorkingCopy>>renameToBe:
    MCWorkingCopy>>requiredPacka
```

```
Monticello-cmm.468
renameToBe: newPackageName
    | newWorkingCopy |
    self modified ifTrue: [ self error: 'Should on
    (MCWorkingCopy allManagers anySatisfy:
        [ :each | each packageName = newPack
    PackageInfo registerPackageName: newPac
    newWorkingCopy := MCWorkingCopy forPac
    self package snapshot definitions do:
        [ :each | each
            rename: self package name
            toBe: newPackageName ].
    self ancestors do:
        [ :each | newWorkingCopy ancestry add.
    ^ newWorkingCopy
```

```
Monticello-cmm.469
renameToBe: newPackageName
    | newWorkingCopy |
    self modified ifTrue: [ self error: 'Should or
    (MCWorkingCopy allManagers anySatisfy:
        [ :each | each packageName = newPac
    PackageInfo registerPackageName: newPac
    newWorkingCopy := (MCWorkingCopy forPa
        repositoryGroup: self repositoryGroup
        requiredPackages: self requiredPackag
        yourself.
    self package snapshot definitions do:
        [ :each | each
            rename: self package name
            toBe: newPackageName ].
    self ancestors do:
        [ :each | newWorkingCopy ancestry adc
    ^ newWorkingCopy
```

Figure 7.9: *Monticello environment: showing the revisions per package (left) – History of one revision & Changes between two revisions: browsing the textual differences of one change (right).*

He was also given access to the tools offered by Monticello to confirm the information he obtained from *JET*. As an experienced developer, the integrator was accustomed to using the Monticello environment shown in Figure 7.9. This figure shows the Monticello tools being used to explore the history of the *Monticello core package* on the Squeak branch (repository located at http://source.squeak.org/trunk). On the left, we see the Monticello tool for browsing the list of revisions per package (*i.e.,* package versions) along with the commit log within a repository. On the right, we see the Monticello tool that allows an user to browse the history of a selected revision (in this case revision `cmm.469`), and the tool that shows the changes between two revisions in that history (in this case between revisions `cmm.468` and `cmm.469`). Along with the changes, a textual view showing the differences introduced by each change is shown.

The tools provided by Monticello have the same purpose as the tools provided by other version control systems (*e.g.,* Subversion). However, with Monticello the information of package versions is presented at the level of classes and methods instead of plain text within files.

7.5.2.1 Protocol

We observed the integrator while he was using *JET* during 5 sessions of 30 minutes each. During these sessions we asked him to talk out loud which made it easier to take notes of his actions. After each session, we asked him for some clarifications about certain choices he made while using the tools. On average, the integrator analyzed 12 deltas per session. According to the integrator, his usual rate for such a task is about 5 to 6 deltas over the same period of time. While this is encouraging, we cannot claim that this speed-up was caused by the use of our tools. It could also have been due to other factors such as the complexity of a change, and so on.

The integrator produced a log for each delta in the list. He wrote a small summary and some notes about the difficulty of integrating each delta and what should be done: if the changes were already integrated, if the changes were applicable to Pharo, if the changes were still valid (not modified later in the list), etc. As an example, the log written for delta `16.nice` is presented below.

```
==============================================================================
Delta: 16.nice (island)

Commit summary:
Use #keys rather than #fasterKeys
Note that pattern (x keys asArray sort) could as well be written (x keys sort) now that keys returns an Array.
This #asArray is here solely for cross-dialect/fork compatibility

Pharo action:
  do nothing - since they reverted the #fasterKeys change to use #keys as we do
==============================================================================
```

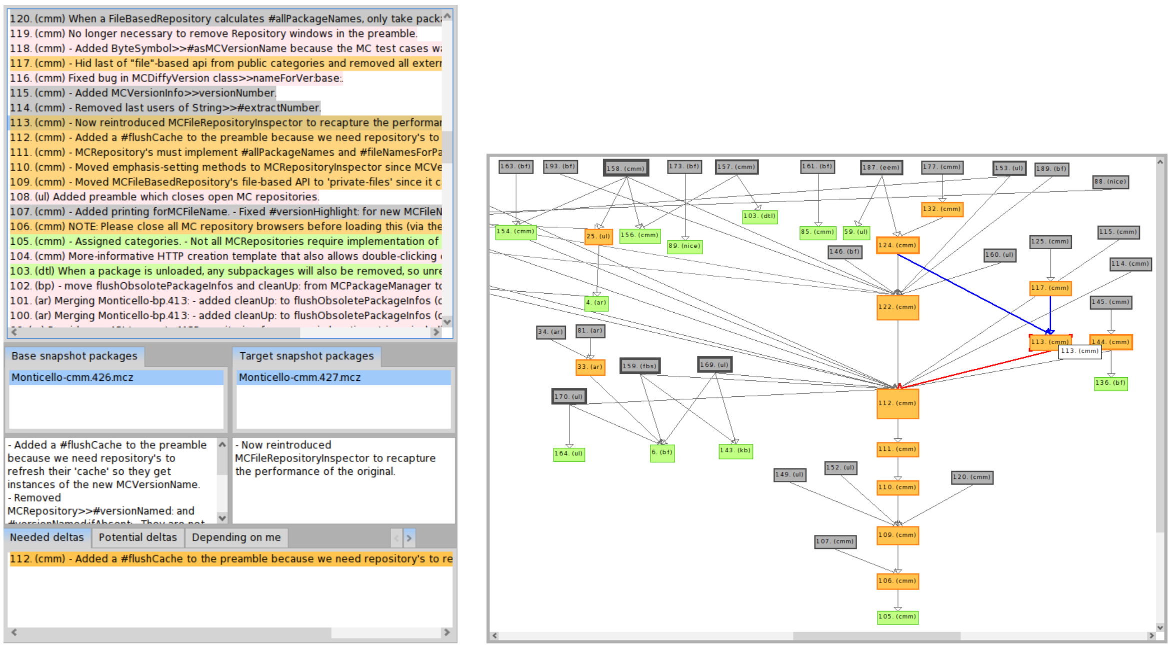
7.5.2.2 Observations

In the following, we describe the observations based on the main actions taken by the integrator.

Identifying committers. The integrator was acquainted with the level of expertise of certain committers. Therefore he took more time to analyze the changes made by not so experienced developers.

From this perspective, having the name[13] of the committer associated with a number for identifying a delta was considered a useful feature of *JET*.

Prioritizing deltas. The integrator started the analysis of the case study by prioritizing the deltas to be integrated based on their complexity. To this end, he based himself on the colors identifying kinds of deltas following the characterization offered by *JET*. This is illustrated in Figure 7.10. On the left, we see the list of all deltas shown on the dashboard (which is equivalent to the list shown in Figure 7.9 on the left). Here the delta `113.cmm` is being explored. On the right, we see the map opened on the same list of deltas. The delta `113.cmm` is highlighting its dependencies and the deltas depending on it by means of the red and blue edges.



```
120. (cmm) When a FileBasedRepository calculates #allPackageNames, only take pack
119. (cmm) No longer necessary to remove Repository windows in the preamble.
118. (cmm) - Added ByteSymbol>>#asMCVersionName because the MC test cases w
117. (cmm) - Hid last of "file"-based api from public categories and removed all exter
116. (cmm) Fixed bug in MCDiffyVersion class>>nameForVer:base:.
115. (cmm) - Added MCVersionInfo>>versionNumber.
114. (cmm) - Removed last users of String>>#extractNumber.
113. (cmm) - Now reintroduced MCFileRepositoryInspector to recapture the performa
112. (cmm) - Added a #flushCache to the preamble because we need repository's to
111. (cmm) - MCRepository's must implement #allPackageNames and #fileNamesForP
110. (cmm) - Moved emphasis-setting methods to MCRepositoryInspector since MCVe
109. (cmm) - Moved MCFileBasedRepository's file-based API to 'private-files' since it c
108. (ul) Added preamble which closes open MC repositories.
107. (cmm) - Added printing forMCFileName. - Fixed #versionHighlight: for new MCFileN
106. (cmm) NOTE: Please close all MC repository browsers before loading this (via the
105. (cmm) - Assigned categories. - Not all MCRepositories require implementation of
104. (cmm) More-informative HTTP creation template that also allows double-clicking
103. (dtl) When a package is unloaded, any subpackages will also be removed, so unre
102. (bp) - move flushObsolotePackageInfos and cleanUp: from MCPackageManager t
101. (ar) Merging Monticello-bp.413: - added cleanUp: to flushObsoletePackageInfos (
100. (ar) Merging Monticello-bp.413: - added cleanUp: to flushObsoletePackageInfos (
```

Base snapshot packages: Monticello-cmm.426.mcz

Target snapshot packages: Monticello-cmm.427.mcz

- Added a #flushCache to the preamble because we need repository's to refresh their 'cache' so they get instances of the new MCVersionName.
- Removed MCRepository>>#versionNamed: and

- Now reintroduced MCFileRepositoryInspector to recapture the performance of the original.

112. (cmm) - Added a #flushCache to the preamble because we need repository's to re

Figure 7.10: *The JET dashboard: a semantically enriched stream of changes (left). The JET map: dependencies between deltas (right).*

The integrator identified that the *islands* (pink) and *sources* (green) deltas were the most suitable candidates to integrate. As *islands* only contain changes without dependencies, he considered these to be easy to integrate and ignored them at the beginning. When asked for the reason, he explained that his motivation was to spend his efforts on the changes that were more complex and thus more challenging to integrate. He started with investigating the *source* deltas in more detail. In particular, he wanted to identify the different 'chains' of deltas within the stream of changes that might constitute a single feature or fix. As such chains originate from a *source* (green), the integrator ignored the *intermediate* (orange) and *end* (grey) deltas for the time being.

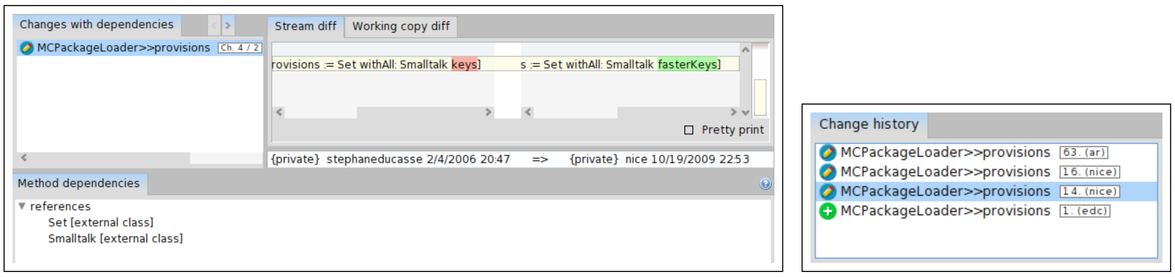
Afterwards, the integrator mentioned that the colors of the nodes were useful in providing an initial assessment of the kinds of deltas, and that the consistent use of the color conventions eased usage of the tool. Despite the presence of the map (see Figure 7.10 on the right), we noticed that the integrator mostly used the dashboard (list view). We hypothesize that this is because the map of the case study was rather complex and the layout algorithm did not succeed in providing an intuitive

[13]In the Squeak community authors are identified by their initials.

layout. Therefore the integrator had to move around nodes (deltas) to get a better understanding of the different chains within the stream. He spent time moving nodes trying to get a mental picture of the dependencies but afterwards mainly used the list.

Using change metrics. The metrics of the changes described in Section 7.3.1 were frequently used by the integrator in combination with the query browser. For example, when he noticed that for a particular change the altered entity was also changed in later deltas, this often served as a cue to open the query browser and inspect the evolution of the changed entity. We identified three different usages where the presence of change metrics supported the integrator.

1. The integrator used the *number of callers* metrics to identify if a particular method should be integrated by checking whether it was called anywhere later on in the stream.
2. The integrator used the *number of changes* metrics to see if methods were still modified later in forthcoming deltas. As a reason, he mentioned that he did not want to integrate changes that would be superseded by other changes.
3. The integrator used the *number of implementors* metrics to see if a changed method was already in use in the current Monticello in Pharo. That was possible because the *JET* tools were loaded in the image in which the integration process was happening.



MCPackageLoader>>provisions Ch. 4 / 2

```
rovisions := Set withAll: Smalltalk keys]        s := Set withAll: Smalltalk fasterKeys]
{private}  stephaneducasse 2/4/2006 20:47    =>    {private}  nice 10/19/2009 22:53
```

▼ references
    Set [external class]
    Smalltalk [external class]

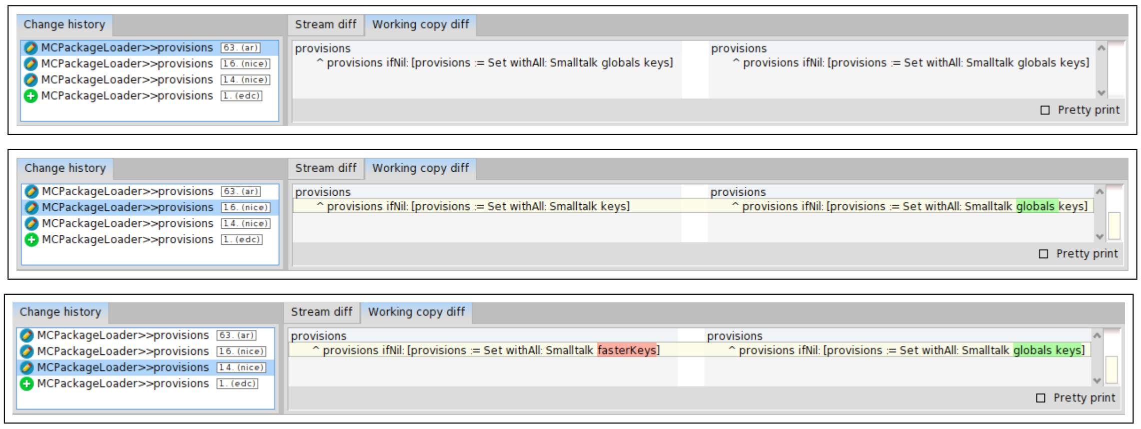
MCPackageLoader>>provisions 63. (ar)
MCPackageLoader>>provisions 16. (nice)
MCPackageLoader>>provisions 14. (nice)
MCPackageLoader>>provisions 1. (edc)

Figure 7.11: *Example: use of* fasterKeys *introduced in delta* `14.nice` *(left) – Changes made to the method* provision *within the stream (right).*

To illustrate this use of *JET*, we briefly discuss an example shown in Figure 7.11 and that it is related to the integrator's aforementioned log. In the history of Squeak, a method fasterKeys was introduced in the implementation of the Dictionary class as an optimized version to return the keys in a dictionary. Consequently, within the Squeak branch of Monticello a method named provisions was changed in delta `14.nice` to make use of this optimized method (as shown in Figure 7.11 – left). The *number of change* metrics for method provision, were `Ch 4/2` meaning that this method changed 4 times in total, of which 2 times in later deltas than the delta in which the use of fasterKeys was introduced (`14.nice`). As the method fasterKeys was not present in Pharo and came from the external reference to class Smalltalk (shown on the diff and dependencies), the integrator was wondering whether this method should also be integrated in order to support the changes made to Monticello in Squeak. By knowing that the provision method was still changed 2 times in later deltas, he was encouraged to first investigate the evolution of the method using the query browser (as shown in Figure 7.11 – right). He found out that this method changed again in deltas `16.nice` and `63.ar`.

As a result, the integrator noticed that the use of fasterKeys was later on reverted in delta `16.nice`, and that in delta `63.ar` the method was changed to reach the same state it was in Pharo (as shown in Figure 7.12 – top). Therefore, these changes could safely be ignored.

Comparing with the current version in Pharo. The final feature of *JET* that the integrator used frequently was the *Working copy diff* to assess the difference between a changed method in the stream and the current version of that method in Pharo. The dashboard and the query browser offer that feature in the source code diff panel.



```
MCPackageLoader>>provisions 63. (ar)
MCPackageLoader>>provisions 16. (nice)
MCPackageLoader>>provisions 14. (nice)
MCPackageLoader>>provisions 1. (edc)
```

```
provisions
    ^ provisions ifNil: [provisions := Set withAll: Smalltalk globals keys]

provisions
    ^ provisions ifNil: [provisions := Set withAll: Smalltalk globals keys]
```

```
provisions
    ^ provisions ifNil: [provisions := Set withAll: Smalltalk keys]

provisions
    ^ provisions ifNil: [provisions := Set withAll: Smalltalk globals keys]
```

```
provisions
    ^ provisions ifNil: [provisions := Set withAll: Smalltalk fasterKeys]

provisions
    ^ provisions ifNil: [provisions := Set withAll: Smalltalk globals keys]
```

Figure 7.12: *Example:* ***working copy diff*** *showing the differences between the changes made to method* provisions *and the version of that method in Pharo.*

In Figure 7.12, we illustrate the use of this feature applied to the previous example. Here we show the differences between the changes to method provision (within the stream) and the version of that method in the working copy (loaded with the current version of provision into Pharo).

Note that the integrator compared the latest version of the method in the stream with its previous and future versions within the stream. The idea was to assess (1) if the change was already in Pharo, and (2) if it was worth to look at this particular version of the change. Note that the previous example also illustrates the usage of this feature.

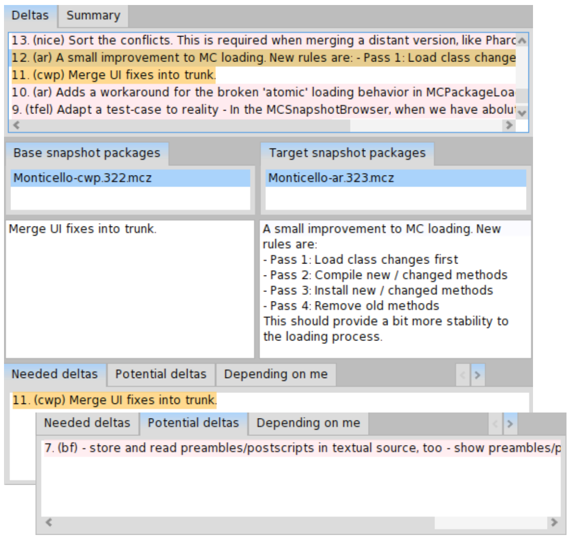
Ignoring potential delta dependencies. Our characterization of *delta dependencies* makes a distinction between *needed* dependencies and *potential* dependencies in order to take the uncertainty introduced by *e.g.,* polymorphism into account.

Figure 7.13 shows both kind of dependencies for the delta `12.ar`. The integrator was confused by *potential* dependencies and decided to ignore them, due to the fact that this introduced quite a few false positives to be processed. In the example shown in Figure 7.13, the potential dependency to delta `7.bf` was indeed due to false positives. Even though *JET* provides support for handling and filtering *potential* dependencies, this is a clear indication that this feature of *JET* should be improved.

7.5.3 Part 2: Effort Estimation by a Developer

7.5.3.1 Protocol

The second part of our case study focuses on providing some insights into the time and effort required to use *JET* to analyze a stream of changes. We asked another *developer* knowledgeable about Mon-



13. (nice) Sort the conflicts. This is required when merging a distant version, like Pharc
12. (ar) A small improvement to MC loading. New rules are: - Pass 1: Load class change
11. (cwp) Merge UI fixes into trunk.
10. (ar) Adds a workaround for the broken 'atomic' loading behavior in MCPackageLoa
9. (tfel) Adapt a test-case to reality - In the MCSnapshotBrowser, when we have abolu

Base snapshot packages: Monticello-cwp.322.mcz

Target snapshot packages: Monticello-ar.323.mcz

Merge UI fixes into trunk.

A small improvement to MC loading. New rules are:
- Pass 1: Load class changes first
- Pass 2: Compile new / changed methods
- Pass 3: Install new / changed methods
- Pass 4: Remove old methods
This should provide a bit more stability to the loading process.

11. (cwp) Merge UI fixes into trunk.

7. (bf) - store and read preambles/postscripts in textual source, too - show preambles/p

Figure 7.13: *Example: **needed** and **potential** dependencies of delta* `12.ar`.

ticello to assess the complete sequence of changes from the Squeak branch of Monticello, and for each delta, determine the potential actions to be taken by someone who wants to integrate that delta into Pharo. More specifically, we asked Marcus Denker to classify the deltas in several categories: *Already integrated* - meaning that the change was already integrated in Pharo; *Ignore* - meaning that it is not relevant or interesting for Pharo; *Unresolved* - meaning that after investigation it is not clear what decision should be taken; and *To integrate* - meaning that the delta is worth integrating and that its impacts are understood and appear to be under control.

For each delta that he processed, we measured the amount of time that he took to assess the delta and decide on its categorization. The developer used a dual-screen setup (27 inch main monitor + 13 inch laptop screen): due to the amount of information provided by *JET* it requires a significant amount of screen real-estate; this setup allowed him to separate the *JET* tools from his code browsing activities. Next to a Pharo image with the *JET* tools loaded, he also had access to the Squeak system in order to explore the context of the original changes. He was left to perform his tasks without interference from the authors.

7.5.3.2 Results

During the time slot of 4 hours that the developer allocated for the case study, he was able to analyze 134 of the 193 deltas. He processed the deltas in chronological order, hence starting with the oldest version (*i.e.,* delta `1.edc`).

Table 7.3 gives a summary of the delta classification made by the developer, along with the total time necessary for the analysis of each group of deltas, and the average amount of time per delta. Next to these average times, we would like to report that there were four *unresolved* deltas that took significantly longer to process than the other deltas (approximately 10 minutes each). Examples of these are the deltas `31.ar` in which trait support was introduced in Monticello, and `154.cmm` in which extensive renaming occurred. As these deltas introduced complex changes to Monticello, it is not surprising that processing them took a relatively long amount of time.

This case study does not provide any claims regarding the correctness of the classification as

	# **deltas**	**total time** **(seconds)**	**∆ average time** **(seconds)**
Already integrated	27	1620	60
Ignore	39	2145	55
To integrate	33	4620	140
Unresolved	35	6300	180
Total	134	14685	–

Table 7.3: *Developer's analysis of the stream of changes: classification of deltas for integration across branches and the time taken by the analysis.*

produced by the developer, but merely serves as a means to analyze the amount of time needed to understand deltas using *JET*. For future work, we plan to use the results provided by the developer and integrate them into Pharo as a means to calculate the number of false positives.

Classification of Deltas. Overall, the developer was able to classify the deltas in a short amount of time. As expected, the cases marked as *Ignore* took little time, as such cases were often features that are either not applicable to Pharo, or that reversed a previous (incorrect) commit. Likewise, cases identified as *Already integrated* were also processed rather quickly. The reason for this, as mentioned by the developer, is that the dashboard includes a view that allows him to compare the difference between a change to an entity and the current version of that entity in Pharo (*i.e.,* usage of the *working copy diff*). Consequently, after a few glances the developer could identify that the changes were already integrated and no further investigation of the delta was needed. After the case study, this led us to believe that such cases could be identified (semi-)automatically, which we consider as a possible extension for a future version of *JET*.

He limited himself to take at the most 10 minutes in analyzing a particular delta, because he preferred to build a larger list of understood deltas than losing time on more complex ones. While the *To Integrate* and *Unresolved* cases took considerably longer to analyze, the amount of time per delta was on average still limited to around 3 minutes. We speculate that this is caused by the fact that the number of cases for which the developer needed to invest a lot of time was rather limited. First, most of the deltas did not contain a lot or complex changes. Second, when the same entity was modified in multiple deltas, the developer had to investigate only one change and then could use the query browser to study the evolution of the entity, resulting in that he had to spend less time analyzing subsequent changes to the same entity. Typical examples of this case are API changes, or reverting to prior changes like the example discussed beforehand. Third, since deltas tend to be related, the developer could spend a considerable amount of time understanding particular deltas; subsequent deltas that were related to this delta were then processed much quicker.

Observations. After the case study, the developer also made a number of observations regarding his process. First, he remarked that the size of the delta is not correlated with the complexity of the analysis required to take a decision. For example, changes to a single polymorphic method could be harder to assess – due to their impact on the system – than a large set of simple changes. Second, the developer remarked that solely analyzing the dependencies of a change did not suffice in order to classify a delta. As one example, he listed the case in which the order of calls in a method was changed. While such a change does not have an impact on the dependencies of the change (as no new calls are added or removed), it can have a drastic impact on the behavior of the system. In such cases, the developer appreciated the presence of the query browser that allowed him to explore the evolution of the method within the stream.

Used Tools. The developer mainly used the dashboard and the query browser. The map was only explored at the beginning of the analysis, where he identified potential complex deltas such as `112.cmm` and `122.cmm`, because multiple deltas were depending on them (as can be seen in Figure 7.10). This confirms that the map is not suitable for performing a in-depth analysis of individual changes but for providing an initial overview of the stream of changes. The metrics provided by each change on the dashboard were heavily used especially at the level of methods. This was combined with the use of the query browser to explore the evolution of methods. Many deltas were quickly assessed because their changes evolved in later deltas, and the developer put more effort on investigating what happened later in the stream.

Analysis of Dependencies. Regarding dependencies, the developer triggered the analysis of each delta by exploring in detail the needed delta dependencies and then quickly exploring the dependencies of a change. In some cases, he just checked the source code diff of a change instead of its change dependencies. This was the case when the textual differences were minimal. However, when the developer was intrigued by the presence of a delta dependency, he explored the changes causing such dependency and their change dependencies. The developer ignored the potential dependencies in most of the cases, and he only paid attention to them when a delta did not have needed delta dependencies. He found out that a good number of these dependencies were false positives. Finally, for deltas where he detected that multiple unrelated changes were present, he took advantage of the dashboard integration with the *Torch* dashboard. In these cases, he first opened the *Torch* dashboard to understand a single delta before studying its dependencies with other deltas within the stream.

7.5.4 Threats to Validity

We performed each evaluation with one participant. Both are core integrators and developers of the Pharo project, and they are knowledgable of the Monticello version control system. They were provided with an explanation of how to use *JET* and its main features.

Both qualitative evaluations allow us to observe how real users interact with the *JET* tools and to estimate the time and effort needed when using *JET* to analyze a concrete stream of changes. The validity of these observations is however subject to a number of threats.

Performed Tasks. One possible threat to validity of the evaluations is that both experiments were performed using the same case study: a stream of changes representing a five-year development of Monticello in the Squeak system. Even though we use one concrete case, it is large and complex enough to illustrate what integrators and developers may deal when analyzing streams of changes and how difficult it can be to determine the dependencies between changes without tool support. In addition, we did not design this case for our evaluations, but we took a real case study that the Pharo integrators wanted to analyze to identify changes that can be cherry picked and integrated into the Monticello system in Pharo.

Participants. One participant for each evaluation might not form a representative sample to evaluate the *JET* tools. However, we are limited by the fact that we need experts of the system to analyze and real integrators that are willing to perform these kinds of experiments. From this perspective, we believe that the results obtained are representative since both participants deal with daily integration tasks and know the Monticello system. Another threat to validate to consider is the bias that existed with the integrator, Stéphane Ducasse, as he is involved in this research. However, he did not participate in the implementation of the tools, and therefore he was not aware of how to use *JET* or what were its features. He had to receive the explanation of how to use the *JET* tools, and then he was observed when analyzing the stream of changes.

7.5.5 Discussion

While both evaluations of *JET* applied to the Monticello case study seem to suggest that *JET* has an added value, the problem of integrating cross-branch changes is not yet solved since we do not do impact analysis and real merging. However, we believe that *JET* is the first step in the direction of such (semi-)automated tool support for assisting integration.

In the following we discuss in more detail both limitations that are already considered as avenues for future work:

Impact of changes. While *JET* provides developers tool support to analyze a stream of changes by offering more information regarding changes such as the characterization of deltas and dependencies within the stream, and by providing navigation and querying facilities of the history of the system, *JET* does not provide guarantees that the code will execute when integrated. It does not provide an impact analysis of the changes. In fact, semantic merging is still a real challenge. This point is even more challenging for dynamically typed languages such as Smalltalk, since static analyses are limited and the code model is less precise. Still tools should be able to show the potential impact that a change may have on the current system. As an avenue for future work, we propose to investigate the use of program slicing or regression testing techniques on both source code and changes to provide a fine-grained impact analysis.

Cross-branch integration. Even though *JET* supports the analysis of streams of changes, it currently does not provide a full-fledged solution for assessing the impact of a stream of changes on a target system and for migrating changes from one branch to another. For example, in our validation the developer performed a change and dependency analysis of the changes made to the Monticello version control system in Squeak without taking into account the evolution of Monticello in Pharo. The developer only looked at the current version of Monticello in Pharo without considering some other versions in its history. Hence, being able to also establish the potential effects of integrating the changes made for Monticello in Squeak into Pharo is required. As an avenue for future work, we propose to extend *JET* such that the history of multiple systems can be taken into account.

7.6 Related Work

To the best of our knowledge, neither related tools nor approaches that aim at understanding commits and assist cherry picking with the goal of merging such commits across branches exist. In the state-of-the-art presented in Section 3.5 several approaches that are related and relevant for our contributions were described. Now that *JET* has been explained, we can provide a more in-depth look of how they differ from *JET*. Moreover, we also introduce other related work in the context of *JET*. In the following, we present approaches that focus on replaying, characterizing, analyzing and understanding changes.

7.6.1 Fine-grained Patching

Semantic patches [Padioleau 2008] offer a declarative domain-specific language (SmPL) for expressing collateral evolutions. With SmPL a developer can describe a generic patch as a transformation of the source code that can be applied to multiple source code files. As an extension of this work spdiff is presented [Andersen 2010]: a tool that, given a set of standard patches, automatically generates a semantic patch.

While semantic patches can be used to generalize a set of changes made in one branch and apply these changes to another branch, they do not fully tackle the issues addressed by *JET*. In particular, semantic patches do not aid in solving the problem that a set of changes might depend on previous changes that were made in the same branch, and that also need to be migrated to obtain a functioning system. As our approach aids integrators in understanding changes and their dependencies within a stream, *JET* is largely complementary to semantic patches and can potentially aid integrators in defining and managing such semantic patches.

Collard *et al.* [Collard 2006] present an approach for easing the integration of large changes by factoring single commits into a series of smaller changes based on syntactic criteria. Based on an XML representation of a *diff* of a system, a developer can partition this *diff* into a number of smaller sets of changes. The idea is that these factored changes can then be integrated individually. First, this process of factoring a commit is done manually and might benefit from the information provided by our tool. Second, similarly to semantic patches, the factored commit does not take into account previous commits and hence does not address the problem of dependencies between changes.

7.6.2 Change Characterization

Dragan *et al.* [Dragan 2011] propose a technique to characterize a commit based on the methods that were added or removed in that commit. In previous work, they have presented a categorization of methods (stereotypes) that take various properties of the method (accessing data, changing state, interaction with other objects,etc.) into account. Their technique leverages these method stereotypes and, by studying the distribution of the various kinds of method stereotypes within a commit, proposes a number of categories of different kinds of commits. This technique is related to ours in the sense that the identified commit types can provide an integrator with valuable information regarding the size and scope of a commit. However, this technique does not provide any information regarding the dependencies between commits and the ease with which a commit can be integrated across branches.

7.6.3 Change Impact Analysis / Change Dependencies

Dependencies between changes have been used in the context of change impact analysis. Chianti [Ren 2004] decomposes the difference between two versions of a Java system into a set of atomic changes. Change impact is then reported in terms of affected (regression or unit) tests whose behavior may have been modified by the applied changes. Chianti relies on syntactic dependencies between atomic changes for the change impact analysis. Other approaches extend Chianti and use dependencies for similar change impact analyses. Ren *et al.* [Ren 2006] extended the syntactic dependencies to three kinds of dependencies between atomic changes that capture syntactic and partially semantic dependencies to detect failure-inducing changes between two versions. While the dependencies provided by Chianti and its derived approaches overlap with our change dependencies, they only apply to a single delta. These approaches do not offer characterization of deltas based on change and delta dependencies within a stream of changes.

CGIs [German 2009] determines the impact of historical code changes on a particular code segment by means of dependence graphs. This approach guides developers to investigate failures in unchanged functions that are affected by bugs introduced in prior code changes. Structural dependencies between C functions are used to build the dependence graphs. These dependencies correspond to a subset of our change dependencies. GENEVA [Herzig 2011] uses dependencies to perform change impact analysis for providing recommendations to developers (*e.g.,* predicting long-term change coupling). This approach builds change dependency graphs (known as change genealogies [Brudaru 2008]) by ordering changes based on dependencies, and later applies model checking to the change genealogy. The dependencies are determined across transactions in version archives. While these dependencies are very similar to our change dependencies, GENEVA's change genealogy and CGIs do not offer the notion of deltas and dependencies between deltas that can be used to characterize sets of changes within a stream. Moreover, both determine dependencies that are not relevant in the context of integration or that can assist in understanding streams of changes and cherry picking.

7.6.4 Understanding Changes

Fritz and Murphy [Fritz 2010] present a study in which they interviewed developers regarding the different kinds of questions they need answered during development. They introduce the information fragment model and associated prototype tool for answering the identified questions. This model provides a representation that correlates various software artifacts (source code, work items, teams, comments, and so on). By browsing the model, developers can find answers to particular development questions.

While a number of the questions that developers need answered during development align with those they need answered during integration of changes, the information fragment model does not provide functionality to calculate dependencies between changes, which is necessary for integrating changes across branches.

The approaches performing change impact analysis presented above provide a means to better understand changes. However, some of them are limited to a single delta, and none of them support understanding streams of changes in the context of integration. *JET* could be complemented with a change impact analysis similar to the one provided by Chianti [Ren 2004].

We introduced *Torch* in Chapter 5. *Torch* is the tool support that we propose as part of our contributions to allow developers understand changes within a single delta. It visualizes how a delta is related to the structure of the system and characterize changes within a delta. *JET* generalizes and augments *Torch*'s philosophy: (1) a stream of changes is characterized, (2) a stream can be queried and navigated, (3) dependencies between the changes are computed and help driving change analyses.

Changes are not treated in isolation but within a stream of changes.

7.7 Conclusion

In this chapter we have presented an approach and (semi-) automated tool support for characterizing a stream of changes. Concretely, deltas, dependencies between changes, and dependencies between deltas are characterized within the stream. Our approach, named *JET*, is part of the requirements of our solution established in Section 2.4 as a means to assist integrators during the integration of changes. *JET* aims at assisting the integration across branches *i.e.,* cherry picking changes from one branch to merge them with another branch.

We have presented five topics in this chapter. First, we explained how deltas and dependencies are characterized. Deltas are characterized based on the presence of dependencies: *source*, *intermediate*, *island* and *end*. Change dependencies are characterized based on the presence of changes on the stream: *local* and *external*, and based on the amount of potential changes existing in the stream that satisfy the dependency: *unique* and *multiple*. Delta dependencies are characterized based on the presence of change dependencies: *needed*, *potential* and *external*.

Second, we have presented the *JET* tools that allow an integrator to visualize and analyze dependencies between changes and deltas themselves. The *dashboard* performs the characterization of changes and provides most of the information to integrators. It offers lists of deltas, lists of changes per delta, lists of dependencies per change, lists of dependencies per delta, and metrics about the amount of changes, deltas and dependencies within the stream. Each change on the list also adds several metrics that extract information from the stream, the history and the working copy. The *map* is a visualization that displays deltas and their dependencies. It provides integrators with a general overview of the complexity of the stream. The *query browser* allows integrators to explore the evolution of an entity within the stream and its users within the stream and within the working copy. It reuses several components of the dashboard such as the list of dependencies per change or the source code diffs.

Third, we discussed how several questions (presented in Section 3.3.2) are supported by *JET*. This part also served as a motivation for the features of *JET* in order to allow answering the questions.

Fourth, we presented the qualitative assessment of the capabilities of *JET* tools by performing an exploratory case study on a considerable five-year stream of changes: changes made to the Monticello version control system in Squeak with the goal of integrating them into Pharo. The evaluation consisted of two parts performed by an integrator and a developer of the Pharo project.

Fifth, we discussed several approaches related to *JET* in the areas of replaying changes, change characterization, change impact analysis, change dependencies and change understanding, and we compared these approaches with *JET*.

CHAPTER 8

Conclusion and Future Work

Contents

8.1 Summary

The motivation of this dissertation is that in a collaborative development environment where the use of branching and therefore merging is extensive, integrators lack adequate support to assist them in performing integration activities such as understanding changes, establishing the dependencies of changes, cherry picking changes, assessing the impact of changes, resolving merging conflicts, and so on.

The complexity of these tasks is aggravated by several factors: (a) the integrators may not be familiar with the code they need to integrate, (b) the changes may come from a branch that has drifted apart from the branch in which the changes need to be integrated (this is common when integrating changes between forks), (c) the support provided by versioning control systems regarding to merging is mostly limited to textual comparisons resulting in conflicts that need to be solved manually, (d) there is no support whatsoever to determine the requirements of a change in order to assist cherry picking, (e) there is no guarantee that after a successful integration the system is 100% functional or that in the future, prior changes do not negatively impact unchanged code.

The integration of changes is key in the development process, however, *developers* that play the role of *integrators* lack adequate support. Note that each of the tasks required to integrate changes may represent a problem by its own, making the whole integration process tedious and time consuming. There is a clear need for tools that can assist integrators in each of these tasks.

In this dissertation we introduced a novel approach that partially tackles the aforementioned problems by supporting assisted integration of changes within a branch and across branches. Our approach is based on an analysis of the integrators' information needs when understanding and integrating changes. This analysis identifies which kinds of information an integrator potentially needs to characterize changes and streams of changes in their respective context. To provide integrators with access to this information, we provide a first-class representation of the history of a system, the changes made to the system, and an analysis of the dependencies between these changes. As a concrete implementation, we presented four meta-models: *Ring*, *RingH*, *RingS* and *RingC*.

On top of these meta-models and analysis, we provide tool support – by means of visualizations and advanced browsers – that allows an integrator to access the information regarding particular changes, and that aids in the decision making process when integrating changes. Two concrete tools have been implemented to support this idea, namely *Torch* that characterizes changes within a single delta and *JET* that characterizes a stream of changes and their dependencies.

8.2 Conclusion

Due to the importance of integration of changes in the software development process and even more in a collaborative development environment, it is necessary to provide integrators with approaches that can assist them in the different activities involved with integration. Throughout this dissertation we have argued that by providing integrators with access to their information needs at an adequate level of abstraction, and tool support to query such information we can assist integration.

Our approach aims at assisting integration as a first step towards supporting full, semantic merging across branches.

First, we performed this research in the context of a real development project – Pharo – and its community, therefore we focused on proposing an approach using scientific methodologies that can be applied in a realistic context. We did not apply our approach to toy examples, but rather to concrete examples of integration problems. We developed our contributions with the intention of integrating them in the core of this project, and we evaluated our contributions by consulting developers and integrators of the Pharo community.

Second, we proposed a language-independent solution that can be used to (semi-)automatically assist integration activities in the context of object-oriented applications. We do not target fully automated support because human expertise is necessary to understand the impact on the semantics of a system. Depending on the context and complexity of the changes it may not be possible to automatically identify impact on semantics. For example, if we intend to integrate a method that impacts the lookup order of a particular call, it is required to have a human check to know if this method can or cannot be integrate.

Third, we presented an initial evaluation of our approach that suggests that our approach and tool support aid integrators in understanding changes in isolation and changes within a stream. Furthermore, we also hypothesize based on our evaluation that our approach can also be used to provide a means to understand changes and stream of changes needed in other contexts. For example: (a) for maintenance tasks where developers may rely on the history of the system to understand the code that they need to change, (b) for aiding new team members to get acquainted with the evolution of the system, and (c) for helping committers to control their changes before committing.

Fourth, despite the fact that our approach only offers a simple change and dependency model along with a dependency analysis, it is able to provide integrators with a significant amount of their information needs. Therefore it can provide answers to most of the questions in the catalogue.

8.3 Integrator Questions Revisited

As a means to show how our approach can assist integrators, we revisit in Table 8.1 the questions from the catalogue introduced in Section 3.3.2 that are supported by our approach. In Section 3.4.5

we already presented two summaries regarding the support provided by our approach in answering these questions. Moreover, in Section 7.4 we discussed how questions related to streams of changes can be supported. Note that in the table for each question we indicate the information required (as described in Section 3.4), the support provided (+ means fully supported and +/- means partially supported) and the tools aiding in answering the question.

Question	**Information required**	**Can be answered?**	**Supporting Tools**
Authorship/Ownership			
A_1	Author/Owner, History	+	*Torch*, *JET*
A_2	Author/Owner, History	+	*Torch*, *JET*
A_3	Author/Owner, History	+	*Torch*, *JET*
A_4	Author/Owner	+	*Torch*, *JET*
A_5	Author/Owner, History	+/-	*Torch*, *JET*
A_6	Author/Owner, History	+	*JET*
Change nature			
B_3	Reason	+/-	*Torch*
B_4	Reason	+/-	*Torch*
B_5	Structure, Change Scope, Kind of Actions, Kind of Entities, Reason	+/-	*Torch*, *JET*
B_6	Kind of Actions, Kind of Entities, Change Dependencies	+/-	*Torch*, *JET*
B_7	Kind of Actions, Kind of Entities, Change Dependencies	+/-	*Torch*, *JET*
B_{14}	Kind of Actions, Kind of Entities, Change Dependencies	+/-	*Torch*, *JET*
Structural change characterization			
S_1	Size, Kind of Entities	+	*Torch*
S_2	Kind of Entities, Structure, Change Scope	+	*Torch*
S_3	Kind of Entities, Structure, Change Scope	+	*Torch*
S_4	Size, Kind of Entities, Structure, Change Scope, Vocabulary, Change Dependencies	+	*Torch*, *JET*
S_5	Kind of Entities, Structure, Change Scope, Kind of Actions, Vocabulary, Reason	+	*Torch*
S_6	Kind of Entities, Structure, Change Scope, Kind of Actions, Reason, Change Dependencies	+	*Torch*, *JET*
S_7	Kind of Entities, Structure, Change Scope, Kind of Actions, Vocabulary, Reason, Change Dependencies	+	*Torch*, *JET*
S_8	Kind of Entities, Kind of Actions, Change Dependencies	+/-	*JET*
S_9	Change Dependencies	+	*JET*
S_{10}	Change Dependencies	+	*JET*
S_{11}	Change Dependencies, Kind of Entities, Structure, Kind of Actions	+	*JET*
S_{12}	Kind of Entities, Structure, Kind of Actions, History, Change Dependencies	+	*JET*
S_{13}	Kind of Entities, Structure, Kind of Actions, Change Dependencies	+/-	*JET*
S_{14}	Kind of Entities, Kind of Actions, History, Change Dependencies	+/-	*JET*
S_{15}	Kind of Entities, Structure, Kind of Actions, History	+	*JET*
S_{16}	Kind of Entities, Kind of Actions, History, Change Dependencies	+/-	*JET*
S_{17}	Vocabulary	+/-	*Torch*, *JET*
S_{18}	Vocabulary	+/-	*Torch*, *JET*
S_{19}	Kind of Entities, Structure, Change Scope, Kind of Actions, History, Change Dependencies	+	*Torch*, *JET*
Bug tracking infrastructure			
I_1	Reason	+/-	*Torch*, *JET*

Continued on Next Page…

Question	Information required	Can be answered?	Supporting Tool
Changes within a stream			
T_1	Time	+	*Torch*, *JET*
T_2	History, Time, Kind of Actions, Kind of Entities	+	*JET*
T_3	History, Kind of Actions, Kind of Entities	+	*JET*
T_4	History, Size, Kind of Actions, Kind of Entities	+	*JET*
T_5	History, Kind of Actions, Kind of Entities	+	*JET*
T_6	History, Time	+	*JET*
T_7	History, Kind of Actions, Kind of Entities	+	*JET*
T_8	History, Kind of Actions, Kind of Entities, Change Dependencies	+	*JET*
T_9	History, Kind of Actions, Kind of Entities, Change Dependencies	+	*JET*
T_{10}	History, Kind of Actions, Kind of Entities	+	*JET*
T_{11}	History, Kind of Actions	+	*JET*
T_{12}	Structure, Kind of Actions, Kind of Entities, Change Dependencies	+	*Torch*, *JET*
T_{13}	History, Time, Author/Owner, Structure, Kind of Actions, Kind of Entities	+	*JET*
T_{14}	History, Kind of Actions, Kind of Entities	+	*JET*
T_{16}	History, Kind of Entities	+	*JET*
T_{17}	History, Kind of Entities	+	*JET*
T_{18}	History, Kind of Entities	+	*JET*
T_{19}	History, Kind of Actions, Kind of Entities	+/-	*JET*
T_{20}	History, Kind of Actions, Change Dependencies	+	*JET*
T_{21}	History, Change Dependencies	+	*JET*
T_{22}	History, Kind of Actions, Kind of Entities, Change Dependencies	+	*JET*
T_{23}	History, Kind of Actions, Kind of Entities	+	*JET*

Table 8.1: *Integrators' questions supported by our approach (+ means fully answered and +/- means partially answered).*

8.4 Limitations and Future Work

In this section we discuss some of the limitations of our approach and propose future work aimed at eliminating these limitations.

8.4.1 Non-Supported Questions

In Table 8.2 we revisit the questions from the catalogue introduced in Section 3.3.2 that are not supported by our approach. At the end we also discuss the questions that are partially answered.

There are 10 out of 64 questions that cannot be answered by our approach. They belong to three categories of questions: *change nature*, *bug tracking infrastructure* and *changes within a stream*. We identified two main reasons of why they are not supported. First, our approach does not take into account information related to other artifacts of the system such as bug reports or tests but it only relies on the source code of a system stored in versioning repositories. Therefore questions such as I_2 and the ones in *change nature* category related to tests (B_8, B_9, B_{10}, and B_{11}) cannot be answered. Second, some questions are very subjective, for example: related to quality (B_1), to correctness (B_2, B_8, B_9, B_{12}, B_{13}) and to renamings (T_{15}).

For future work, our approach can support answering the first kind of questions by incorporating

Id	Question
Change nature	
B_1	Does this change improve the quality?
B_2	Is this change correct?
B_8	Did this change fix/break tests? Which tests?
B_9	Did the tests work before the changes?
B_{10}	How can I test this change?
B_{11}	Is the change covered by tests? What is the coverage?
B_{12}	If I just apply the change, what are the parts of my current system that it will break?
B_{13}	If the merge succeeds, will the change work later?
Bug tracking infrastructure	
I_2	Have other bugs related to the change been reported?
Changes within a stream	
T_{15}	Was this method/class renamed in the past? in which version?

Table 8.2: *Non-supported questions*

other sources of information, such as bug trackers or tests. To support the second category of questions that regard more semantical information, our approach can be completed with metrics/link rules to answer questions related to quality, with unit testing to answer questions related to correctness, and with the use of refactoring tools to answer behavioral preserving questions such as renamings.

Finally, as seen in Table 8.1 there are 15 questions that can be partially answered (+/-). Six questions are in the *change nature* category, six questions are in the *structural change characterization* category, and one question in the categories *bug tracking infrastructure*, *authorship/ownership* and *change within a stream*. Most of these questions are partially answered because of the same reasons mentioned before. Other questions such as “What is the total impact of this change?” (B_6) are only partially supported because our approach does not provide a change impact analysis (which is discussed later).

8.4.2 Improvements

Our discussion of future work is based on several improvements of the work we have presented and in particular of the *Torch* and *JET* tools.

Simultaneous analysis of multiple branches

Our approach supports the analysis of a stream of changes from a *source* branch in order to cherry picking changes that can be integrated with a *target* branch. However, within *JET* we do not fully provide an analysis that takes the evolution of both branches into account at the same time as a means to identify the implications of merging such changes. Currently, our approach considers the stream of changes from the *source* branch and the current version of the *target* branch. This is not enough to completely assess the impact of integrating a stream of changes into a *target* branch or to identify what can be migrated from one branch into another without interfering with past integration actions. For example, consider that the stream may be reintroducing features that were previously removed from the target branch.

For supporting the analysis of multiple branches we should take into account that our change and dependency model may need to be extended with other representations *e.g.,* cross dependencies, the importers may need adjustments and our tools *Torch* and *JET* can be still applied but as part of a tool suite that incorporates new tool support. The concrete idea is to provide tool support for performing a cross-branch analysis where both streams of changes are compared and can be explored within the same (visual) dashboard, and see for instance that a change (fix) in the *source* branch was applied to the *target* branch, and therefore it is actually redundant, even though, in the *target* branch the problem was fixed in a different way (making visible such differences between both fixes).

Cross-branch integration

Our methodology and tool support provide a means to comprehend changes and stream of changes in isolation. That means our approach is used to make integration decisions and later using the version control system perform the actual integration. In order to provide comprehensive support for integration our approach should also provide a means to allow integrators to cherry pick changes and merge them with the target branch from within our tools, making the process smooth for integrators. This is considering that our approach provides detailed information about the changes that can ease and speed up the integration. For example, an integrator may want to merge a change and take its dependencies into account automatically from within the dashboards. To achieve this, our tools *Torch* and *JET* need to be extended with version control facilitates. In fact, *Torch* and *JET* can be part of a new generation of version control systems.

Change impact analysis

We presented several approaches that propose change impact analysis in Sections 3.5.3 and 7.6. Most of them are related to our approach because they use the notion of dependencies between changes although limited to pairs of versions. We put that our approach can be enhanced with an explicit impact analysis to (semi-)automatically infer the impact of merging a single change within a branch or a stream of changes across branches, concretely, to find out whether the semantics of the system is affected if integrating a particular change. For example, if a merge adds an overridden method in a class hierarchy, our approach and tools could identify *if* and *how* this change impacts the semantics. This could allow us to give the integrator warnings when the semantics is affected, in this example something like "merging this change will have an impact on what is executed in the super call".

These approaches can serve as the foundation to define a change impact analysis that can be incorporated in our approach. Even though most of them also rely on test prioritization, we already mentioned that our approach can be complemented with unit testing to support non-answered questions. Such source of information can be used for impact analysis as well. Another point to take into account is the use of Reuse Contracts [Steyaert 1996] as a means to perform behavioral analysis

8.4.3 Full-fledged Validation

In Chapters 5 and 7 we presented the evaluations that were performed to assess our contributions. Even though, we executed two qualitative evaluations using *JET* for the analysis of a five-year stream of changes of a real system, we cannot make any generalizable claims regarding the usefulness of *JET* for characterizing streams of changes and assisting cherry picking. We consider it necessary to perform a controlled experiment with advanced developers and integrators to assess *JET*, its characterization of deltas and dependencies, and its integration with *Torch*. For this, we intend to reach integrators and developers from the Smalltalk community, in particular from the Pharo and ESUG communities. However, we are aware that getting them all together is unfeasible and therefore, we may consider to travel to various research institutions and companies that use Smalltalk to perform individual evaluations of the use of our approach.

Applying our approach and tools in multiple integration scenarios provide us with means to assess the scalability of our approach and allows us to identify the real benefits of our approach in assisting the integration process. Moreover, by means of a more elaborated evaluation we can identify other improvements and refinements of our tools and formalism.

8.4.4 Other Improvements

For future work we have already identified several improvements to our tools that do not require major engineering efforts or the addition of new information.

First, we propose to enhance *Torch* with some features of *JET* and vice versa. We can incorporate the notion of dependencies within *Torch* and provide a characterization of dependencies similar to the one in *JET*. This can allow us to directly infer co-related changes within a delta. Moreover, *RingS* can also represent method's relationships (method calls, class references and variable access) as is done in *RingC* to provide finer-grained information on the *Torch* dashboard and visualizations. In *JET* we can incorporate visualizations similar to the ones in *Torch* for exploring deltas together with its changes, or for providing a visual dashboard as the one in *Torch*.

Second, we also consider several improvements for *Torch* that were discussed in Sections 5.7.1 and 5.7.4. Most of them correspond to technical adjustments to customize the visualizations within the dashboard. Another suggestion is related to the classification of changes in the visualizations by their semantic impact. We can achieve this by providing the change impact analysis discussed above.

8.5 Contributions

We first present a summary of our contributions in Table 8.3, and then we conclude by describing the conceptual and technical contributions of the work we have presented in this dissertation to tackle an important problem existing in the software development process, namely the lack of support for assisted integration of changes within a branch and across branches.

8.5.1 Conceptual Contributions

- We present an in-depth analysis of the problem underlying this research. We explain the relevance of collaborative software development, how version control systems support this kind of development by means of branching and merging, the challenges that need to be faced to support integration, and the requirements of a solution that should be considered to assist integration.

- We present a catalogue of 64 questions that shows which questions integrators ask themselves when performing integration activities. We gathered most of the questions from a study in

	Supporting Integration Activities	
	within system (two versions)	**across branches** (sequence of versions)
Models	○ *Ring* ○ *RingS*	○ *Ring* ○ *RingH* ○ *RingC*
Study		○ catalogue of questions (64)
Analyses	◇ generation of changes ◇ characterization of changes	◇ generation of history ◇ calculation of deltas ◇ calculation of dependencies ◇ characterization of dependencies ◇ characterization of deltas
Tool Support	*Torch*	*JET*
Evaluation	▷ questionnaire (6 integrators) ▷ pre / post tests (10 developers) ▷ examples	▷ qualitative (integrator) ▷ qualitative (developer) ▷ examples

Table 8.3: *Summary of contributions*

which 20 developers that integrate changes within Smalltalk projects participated. Moreover, we complemented their questions with other questions regarding integration that we found in related studies. The goals of this study were to identify the integrators' information needs to establish our requirements, and to use these questions as a means to validate our contributions.

- We present a characterization of integrators' information needs based on the catalogue of questions. The identified kinds of information were classified as (a) descriptive information (*e.g.,* size), (b) structural information (*e.g.,* change scope), (c) semantic information (*e.g.,* reason), and (d) historical information (*e.g.,* change dependencies). By means of this we aim at providing integrators with the required information to assist the integration.

- We propose a first-class representation of the history and changes of a software system stored in versioning repositories. This serves as the underlying representation for history, version comparisons and streams of changes analyses (*i.e.,* delta and dependency analyses) to characterize changes and streams of changes.

- We propose the idea of assisted integration by means of the characterizations of changes and streams of changes, together with (semi-)automated tool support that can be used by integrators to access their information needs.

- We propose the use of visualizations as a way to observe patterns in the changes (*e.g.,* removal of features, refactorings, etc.). By means of visual representations of the changes within their context our approach aims at providing an overview of these changes, easing and speeding up the comprehension of these changes.

- We performed several evaluations of our approach: First, regarding the assisted integration of a single delta within a branch we presented: (a) a questionnaire intended for integrators in the Smalltalk community to evaluate the characterization of changes within a single delta and our tool support (*Torch*), (b) a pre-test and a post-test intended for developers to identify what support they require to understand and integrate changes, and to evaluate how the characterization of changes provided by *Torch* helped them, and (c) examples showing how *Torch* can ease in understanding changes made to the Pharo environment. Second, regarding the assisted integration of a stream of changes across branches we presented: (a) a qualitative evaluation with a Pharo integrator that analyzed a considerable stream of changes made to fork (*i.e.,* Squeak) using our tool support (*JET*), (b) a qualitative evaluation with a Pharo developer that analyzed the same stream of changes using *JET* and categorized these changes (*e.g.,* integrate, ignore, ..), and (c) examples showing how *JET* can assist integrators answering the questions they raise during the integration.

- We propose an approach that is general enough to assist integration activities such as understanding changes, cherry picking changes, assessing changes, etc. in any software development process. Our approach is inherently language independent and it can be applied to assist integrators in a collaborative development environment independently of the programming language and infrastructure used.

8.5.2 Technical Contributions

- We implemented the *Ring* source code meta-model. This meta-model is not only the foundation of our other meta-models *RingS*, *RingH* and *RingC*, but it is also used as the Smalltalk source code meta-model to support tool integration in the Pharo environment. *Ring* has been adopted by the Pharo community and it is integrated in *Pharo 1.4* which was released in April 2012.

- We implemented the *RingS* change meta-model to define models that denote the difference between two versions. *RingS* is built on top of the *Ring* source code meta-model. We refer to *RingS* as the single-delta change model regarding the fact that we compare two versions within the system (*i.e., base* and *target* version). However, *RingS* differs from other change models in the sense that it not only defines the changes between both versions, but it defines the complete state of the *target* version and what changed from *base* to *target*. By means of this we can offer integrators the context in which the changes occurred.

- We implemented the *RingH* history meta-model to define history models containing the whole or partial evolution of a system. This model is built on top of the *Ring* source code meta-model and therefore provides representations of language constructs in Smalltalk. Despite this, *RingH* is generic enough to be extended to support program definitions in other object-oriented programming languages such as interfaces in Java.

- We implemented the *RingC* change meta-model to define models containing a stream of changes in the form of deltas, changes (*i.e.,* additions, modifications and removals) and dependencies. *RingC* is built on top of *Ring* and it uses *RingH* because a change model is derived from a history model. A change model consists of the changes made to the program entities and relationships that exist within the history model.

- We implemented the *Torch* tool to provide the characterization of changes within single deltas. *Torch* is the client of the *RingS* meta-model and it allows integrators to visually comprehend changes and their context using descriptive, structural and semantic information. *Torch* offers integrators a dashboard that gathers as much information as possible regarding the changes. It includes several visualizations, fly-by-helps, diffs, metrics, summaries and lists of changes to ease the navigation and querying of the changes. Moreover, we have integrated *Torch* with the Monticello version control system to ease the access of the source code repositories. We have also enabled *Torch* to support *RingH* models as the input for defining *RingS* models.

- We implemented the *JET* tools to provide the characterization of streams of changes. *JET* is the client of the *RingC* meta-model, which in turn uses a *RingH* model. *JET* extends *Torch*'s philosophy but offering integrators a means to comprehend a stream of changes, their context and their dependencies using descriptive, structural, semantic and historical information. *JET* offers three tools: (1) the dashboard that characterizes the stream and allows integrators to explore deltas, changes and dependencies along with metrics and diffs, (2) the map that offers a visual overview of the dependencies between deltas, and (3) the query browser that provides a means to explore the evolution of an entity within the stream. Furthermore, we have integrated *JET* with *Torch* to allow integrators to perform an in-depth analysis of any delta within the stream.

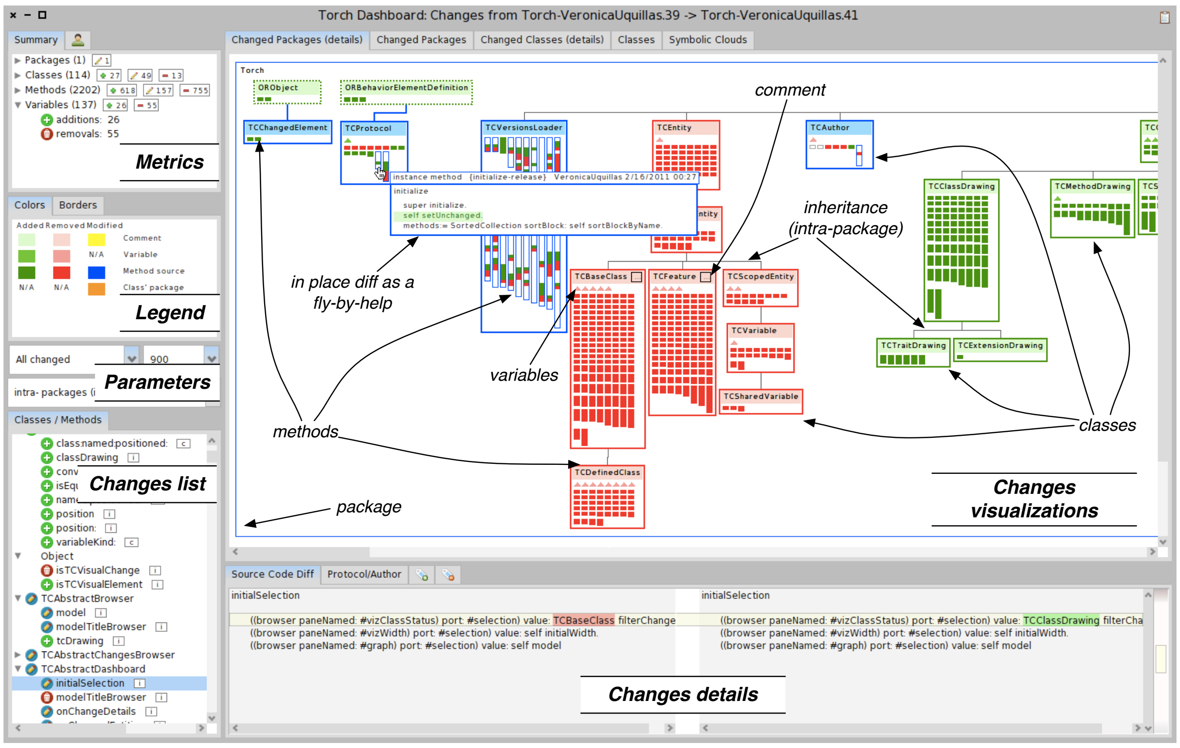
APPENDIX A

Torch: Field Evaluation Questionnaire

1	2	3	4	5
Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree

	1	2	3	4	5
Qualify the context					
Is software change integration difficult?					
Qualify yourself					
Are you skilled using visualizations?					
Do you find visualizations useful?					
Are you an integrator?					
Are you an expert of the system(s) that you integrate?					
Common tasks when comparing versions					
identify if it contains one fix or multiple fixes?					
identify / characterize changes (semantic fix, cosmetic/doc, structural/organizational fix) or (maintenance, feature addition/removal, enhancements...) ?					
assess criticality of changes?					
analyze the impact of changes?					
compare branches for merge?					
Torch experience					
"Imagine that Torch would be more than a prototype" Would you like to use Torch in your daily integration process?					
Does the System Dashboard help you?					
Does the Package Dashboard help you?					
Did you use the Enhanced Changes List? was it useful?					
Do you understand the two different scenarios (changes against ancestor and working copy) in the MC Repository browser?					
Did you use one of those scenarios more frequently? which one?					
Do you like the fact that we do not show only visual elements but also the Diff tool all the time?					
Do you find the fly-by help showing code on any entity useful?					
Do you think that you got a better understanding of the changes, their scope and their impact using Torch instead of the tools you currently use?					

For the components of the Torch Dashboard:



instance method {initialize-release} VeronicaUquillas 2/16/2011 00:27

```
initialize

   super initialize.
   self setUnchanged.
   methods:= SortedCollection sortBlock: self sortBlockByName.
```

```
initialSelection

   ((browser paneNamed: #vizClassStatus) port: #selection) value: TCBaseClass filterChange
   ((browser paneNamed: #vizWidth) port: #selection) value: self initialWidth.
   ((browser paneNamed: #graph) port: #selection) value: self model
```

```
initialSelection

   ((browser paneNamed: #vizClassStatus) port: #selection) value: TCClassDrawing filterCha
   ((browser paneNamed: #vizWidth) port: #selection) value: self initialWidth.
   ((browser paneNamed: #graph) port: #selection) value: self model
```

	have used		considered useful				
	Yes	No	1	2	3	4	5
Components							
Summary (metrics & users)							
Parameters (class state & width)							
Changes List							
Diff Tool (& other data)							
Visualizations							
Changed Packages (details)							
Changes Packages (condensed)							
Packages (condensed)							
Changed Classes (details)							
Classes (condensed)							
Symbolic Clouds							
Entity Names							
Navigation Features							
Class Structure fly-by help							
Source Code fly-by help							
Contextual Menu							
Changes List refreshing by selected entity or metric							

Continued on Next Page…

	have used		considered useful				
	Yes	No	1	2	3	4	5
Legends							
Colors (kinds of changes and entities)							
Border Styles (kinds of changes and entities)							
Line Styles (kinds of relations)							

Open questions

- Which applications do you commonly use for version comparison (e.g. Changes List, Monticello Changes List, Diff File tools, etc.)?
- Which software projects do you mostly integrate?
- Which features of Torch need to be improved?
- Which features of Torch are not useful at all and should be removed? Why?
- Do you think there exist some aspects that are not covered by Torch? Which features are missing? What do you recommend us to add?
- Do you know about existing approaches/tools intended for version comparison presenting (visualizations with) the structural model and changes as Torch does? If yes, mention them.

APPENDIX B
Torch: User Study Pre-Test

Personal Background

Please answer the following questions with regard to your age and education background. The answers will be kept private and only serve to put your other answers in context.

What is your age?:

Please sketch your background (PhD, Master, Bachelor, etc):

Questionnaire

For each of the statements below, please rate each one on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extend they represent your opinion.

1	**2**	**3**	**4**	**5**
Totally disagree	Disagree	Neither agree nor disagree	Agree	Totally agree

	Scale				
General Background					
I consider myself an experienced developer	1	2	3	4	5
I consider myself a proficient OO developer	1	2	3	4	5
I am an expert user of the Pharo or any other Smalltalk IDE	1	2	3	4	5
I use facilities (search, senders, implementors, . . .) of my IDE when coding	1	2	3	4	5
I use facilities (change lists, merge) of my version control system	1	2	3	4	5
Attitude towards tool support					
IDEs are essential for software development	1	2	3	4	5
Version Control Systems (e.g. Git, Monticello, Store) are essential for software development	1	2	3	4	5
I find software visualizations useful for understanding source code	1	2	3	4	5

Continued on Next Page. . .

	Scale				
I find diagrams of my software (UML, ...) useful	1	2	3	4	5
Attitude towards change understanding					
Advanced development tools ease software development	1	2	3	4	5
Understanding changes of the developers is an essential part of the development process	1	2	3	4	5
Understanding changes can be difficult when you did not implement them	1	2	3	4	5
Version control systems provide sufficient help in understanding the changes of a program	1	2	3	4	5
Having a visualization helps understanding changes	1	2	3	4	5
Most questions about source code changes can be answered using the facilities of versioning control systems	1	2	3	4	5
Expectations of a change visualization tool					
Visually exploring changes is essential for change understanding	1	2	3	4	5
Graphical notations should make it easier to identify changes than just reading code	1	2	3	4	5
Visualizations should provide semantic information about changes	1	2	3	4	5
Graphical and textual information about changes should be combined together to speed change exploration and understanding	1	2	3	4	5
The Pharo IDE provides sufficient change understanding facilities	1	2	3	4	5
Monticello provides sufficient change understanding facilities	1	2	3	4	5

APPENDIX C

Torch: User Study Post-Test

Questionnaire

For each of the statements below, please rate each one on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extend they represent your opinion.

1	**2**	**3**	**4**	**5**
Totally disagree	Disagree	Neither agree nor disagree	Agree	Totally agree

	Scale				
Exercise Experience					
I was able to follow and complete the exercise	1	2	3	4	5
The exercise was too simple	1	2	3	4	5
The tasks of the exercise are the same to the ones I perform during development	1	2	3	4	5
Torch Experience					
The Torch package-based visualizations provide an overview of changes	1	2	3	4	5
I find the Torch package-based visualizations useful to understand changes	1	2	3	4	5
I find the Torch class-based visualizations useful to understand changes	1	2	3	4	5
I find the Torch symbolic clouds useful to understand changes	1	2	3	4	5
Visual representation of program entities in Torch are easy to understand	1	2	3	4	5
Evaluation of Torch					
Torch's use of class representations (UML like) makes change identification easier	1	2	3	4	5
Torch's use of source code as a fly-by help makes change exploration easier	1	2	3	4	5
Torch brings semantic information not provided by textual change lists	1	2	3	4	5
Torch complements Monticello's change exploration facilities	1	2	3	4	5
Visualizations in Torch are easy to understand	1	2	3	4	5

Continued on Next Page...

	Scale				
Torch aids in understanding changes	1	2	3	4	5
I would use Torch in my development activities	1	2	3	4	5
Features of Torch					
I find the detailed class representation useful	1	2	3	4	5
I find the condensed class representation unnecessary	1	2	3	4	5
The package-based visualizations provide enough information to get an idea of the changes	1	2	3	4	5
I like the fly-by help to explore the source code from within the visualizations at any time without opening a new window	1	2	3	4	5
I like the fly-by help to explore the complete structure of any class	1	2	3	4	5
I find the mixed symbolic cloud redundant	1	2	3	4	5
I find the presence of unchanged program entities needed to understand the context of the changed ones	1	2	3	4	5

Comments

Please note any additional comment you might have, either on the assignment, or on the tool. Please include features that you think might be enhanced, or added in order to make Torch a better tool (You can continue on the other side of the sheet).

Bibliography

[Abdi 2006] M. K. Abdi, H. Lounis and H. Sahraoui. *Analyzing Change Impact in Object-Oriented Systems*. In Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, EUROMICRO'06, pages 310–319, 2006. IEEE Computer Society.

[Abdi 2009] Mustapha Abdi, Hakim Lounis and Houari Sahraoui. *Analyse et prédiction de l'impact de changements dans un système à objets : Approche probabiliste*. In Proceedings of LMO'09, 2009.

[Abebe 2009] Surafel Lemma Abebe, Sonia Haiduc, Andrian Marcus, Paolo Tonella and Giuliano Antoniol. *Analyzing the Evolution of the Source Code Vocabulary*. In Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, CSMR'09, pages 189–198. IEEE Computer Society, 2009.

[Adams 1986] Evan Adams, Wayne Gramlich, Steven S. Muchnick and Soren Tirfing. *SunPro: engineering a practical program development environment*. In International workshop on Advanced programming environments, pages 86–96, 1986. Springer-Verlag.

[Alam 2009] Omar Alam, Bram Adams and Ahmed E. Hassan. *Measuring the progress of projects using the time dependence of code changes*. In Proceedings of the 25th IEEE International Conference on Software Maintenance, ICSM'09, pages 329–338. IEEE, 2009.

[AmcomTechnology 2010] AmcomTechnology. *Source Code Management with Multiple QA environments using Git*. http://www.amcomtech.net/client/index.cfm/2010/10/28/Source-Code-Management-with-Multiple-QA-environments-using-Git, 2010. [Online; accessed 30-May-2012].

[Andersen 2010] J. Andersen and J. Lawall. *Generic patch inference*. Automated Software Engineering, vol. 17, no. 2, pages 119–148, 2010.

[Anderson 2001] Paul Anderson and Tim Teitelbaum. *Software Inspection Using CodeSurfer*. In Proceedings of the International Workshop on Inspection in Software Engineering, WISE'01, 2001.

[Anquetil 1998] Nicolas Anquetil and Timothy C. Lethbridge. *Assessing the relevance of identifier names in a legacy software system*. In Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, CASCON'98, pages 213–222. IBM Press, 1998.

[Apel 2011] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer and Christian Kästner. *Semistructured merge: rethinking merge in revision control systems*. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE'11, pages 190–200. ACM, 2011.

[Arbuckle 2008] Tom Arbuckle. *Visually Summarising Software Change*. In Proceedings of the 12th International Conference Information Visualisation, pages 559–568. IEEE Computer Society, 2008.

[Asklund 1994] Ulf Asklund. *Identifying Conflicts During Structural Merge*. In Nordic Workshop Programming Environment Research, pages 231–242, 1994.

[Badri 2005] L. Badri, M. Badri and D. St-Yves. *Supporting predictive change impact analysis: a control call graph based technique*. In Proceedings of the 12th Asia-Pacific Software Engineering Conference, APSEC'05, pages 9–15, dec 2005.

[Bergel 2005] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz and Roel Wuyts. *Classboxes: Controlling Visibility of Class Extensions*. Journal of Computer Languages, Systems and Structures, vol. 31, no. 3-4, pages 107–126, December 2005.

[Berlage 1993] Thomas Berlage and Andreas Genau. *A framework for shared applications with a replicated architecture*. In Proceedings of User Interface Software and Technology Symposium, UIST'93, pages 249–257, 1993. ACM.

[Berlin 2006] Daniel Berlin and Garrett Rooney. Practical Subversion, second edition. Apress, 2006.

[Berliner 1990] Brian Berliner. *CVS II: Parallelizing Software Development*. In Proceedings of the USENIX Conference (The Advanced Computing Systems Professional and Technical Association), pages 22–26, 1990.

[Berzins 1994] Valdis Berzins. *Software merge: semantics of combining changes to programs*. Journal of ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 16, no. 6, pages 1875–1903, 1994.

[Beyer 2005] Dirk Beyer. *Co-change visualization*. In Proceedings of the 21st IEEE International Conference on Software Maintenance, Industrial and Tool volume, ICSM'05, pages 89–92, 2005.

[Binkley 1995] David Binkley, Susan Horwitz and Thomas Reps. *Program integration for languages with procedure calls*. Journal of ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 4, no. 1, pages 3–35, 1995.

[Bird 2011] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall and Premkumar Devanbu. *Don't touch my code!: examining the effects of ownership on software quality*. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE'11, pages 4–14. ACM, 2011.

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Bohner 1996] Shawn A. Bohner and Robert S. Arnold. Software change impact analysis. IEEE Computer Society Press, 1996.

[Brant 1998] John Brant and Don Roberts. *"Good Enough" Analysis for Refactoring*. In Object-Oriented Technology Ecoop '98 Workshop Reader, LNCS, pages 81–82. Springer-Verlag, 1998.

[Brudaru 2008] Irina Ioana Brudaru and Andreas Zeller. *What is the long-term impact of changes?* In Proceedings of the 2008 international workshop on Recommendation Systems for Software Engineering, RSSE'08, pages 30–32. ACM, 2008.

[Buffenbarger 1995] Jim Buffenbarger. *Syntactic Software Merging*. In Selected papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management, pages 153–172, 1995. Springer-Verlag.

[Bunge 2009] Philipp Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, April 2009.

[Ceri 1989] S. Ceri, G. Gottlob and L. Tanca. *What You Always Wanted to Know About Datalog (And Never Dared to Ask)*. IEEE Transactions on Knowledge and Data Engineering, vol. 1, no. 1, pages 146–166, 1989.

[Chacon 2008] Scott Chacon. *Git Internal*. PeepCode, 2008.

[Chaumun 2002] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller and Francois Lustman. *A change impact model for changeability assessment in object-oriented software systems*. Science of Computer Programming, vol. 45, no. 2-3, pages 155 – 174, 2002.

[Chawathe 1996] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina and Jennifer Widom. *Change detection in hierarchically structured information*. pages 493–504, 1996.

[Chesley 2005] Ophelia C. Chesley, Xiaoxia Ren and Barbara G. Ryder. *Crisp: A Debugging Tool for Java Programs*. In Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM'05, pages 401–410, 2005.

[Coelho 2006] Wesley Coelho and Gail C. Murphy. *Presenting crosscutting structure with active models*. In Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD'06, pages 158–168, 2006. ACM Press.

[Collard 2006] M. Collard, H. Kagdi and J. Maletic. *Factoring Differences for Iterative Change Management*. In International Workshop on Source Code Analysis and Manipulation, SCAM'06, pages 217–226. IEEE, 2006.

[Collins-Sussman 2009] Ben Collins-Sussman, Brian W. Fitzpatrick and C. Michael Pilato. Version control with subversion (for subversion 1.6). O'Reilly Media, June 2009.

[D'Ambros 2006] Marco D'Ambros, Michele Lanza and Mircea Lungu. *The Evolution Radar: Integrating Fine-grained and Coarse-grained Logical Coupling Information*. In Proceedings of the 3rd International Workshop on Mining Software Repositories, MSR'06, pages 26 – 32, 2006.

[D'Ambros 2007] Marco D'Ambros and Michele Lanza. *BugCrawler: Visualizing Evolving Software Systems*. In Proceedings of the 11th IEEE European Conference on Software Maintenance and Reengineering, CSMR'07, page to be published, 2007.

[D'Ambros 2008] M. D'Ambros, H. Gall, M. Lanza and M. Pinzger. *Analysing Software Repositories to Understand Software Evolution*. In Software Evolution, pages 37–67. Springer-Verlag, 2008.

[de Moor 2007] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni and J. Tibble. *Keynote address: .QL for source code analysis*. In IEEE Computer Society, editeur, Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM'07, pages 3–16, 2007.

[De Volder 2006] K. De Volder. *JQuery: A Generic Code Browser with a Declarative Configuration Language*. Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages, pages 88–102. Springer, 2006.

[Demeyer 2001] Serge Demeyer, Sander Tichelaar and Stéphane Ducasse. *FAMIX 2.1 — The FAMOOS Information Exchange Model*. Rapport technique, University of Bern, 2001.

[Dig 2008] Danny Dig, Kashif Manzoor, Ralph E. Johnson and Tien N. Nguyen. *Effective Software Merging in the Presence of Object-Oriented Refactorings*. IEEE Transactions on Software Engineering, vol. 34, no. 3, pages 321–335, 2008.

[Dragan 2006] Natalia Dragan, Michael L. Collard and Jonathan I. Maletic. *Reverse Engineering Method Stereotypes*. In Proceedings of the 22nd IEEE International Conference on Software Maintenance, ICSM'06, pages 24–34. IEEE, 2006.

[Dragan 2009] Natalia Dragan, Michael L. Collard and Jonathan I. Maletic. *Using method stereotype distribution as a signature descriptor for software systems*. In Proceedings of the 25th IEEE International Conference on Software Maintenance, ICSM'09, pages 567–570. IEEE, 2009.

[Dragan 2010] Natalia Dragan, Michael L. Collard and Jonathan I. Maletic. *Automatic identification of class stereotypes*. In Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM'10, pages 1–10. IEEE, 2010.

[Dragan 2011] N. Dragan, M. Collard, M. Hammad and J. Maletic. *Categorizing Commits Based on Method Stereotypes*. In Proceedings of the 27th IEEE International Conference on Software Maintenance, ICSM'11, pages 520–523. IEEE, 2011.

[Ducasse 1999] Stéphane Ducasse. *Evaluating Message Passing Control Techniques in Smalltalk*. Journal of Object-Oriented Programming (JOOP), vol. 12, no. 6, pages 39–44, June 1999.

[Ducasse 2000] Stéphane Ducasse, Michele Lanza and Sander Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools, CoSET '00, June 2000.

[Ducasse 2005] Stéphane Ducasse and Michele Lanza. *The Class Blueprint: Visually Supporting the Understanding of Classes*. Transactions on Software Engineering (TSE), vol. 31, no. 1, pages 75–90, January 2005.

[Ducasse 2006a] Stéphane Ducasse, Tudor Gîrba and Adrian Kuhn. *Distribution Map*. In Proceedings of 22nd IEEE International Conference on Software Maintenance, ICSM'06, pages 203–212, 2006. IEEE Computer Society.

[Ducasse 2006b] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts and Andrew P. Black. *Traits: A Mechanism for fine-grained Reuse*. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 28, no. 2, pages 331–388, March 2006.

[Ducasse 2009] Stéphane Ducasse, Marcus Denker and Adrian Lienhard. *Evolving a Reflective Language*. In Proceedings of the International Workshop on Smalltalk Technologies, IWST'09, pages 82–86, aug 2009. ACM.

[Ducasse 2010] Stéphane Ducasse, Lukas Renggli, C. David Shaffer, Rick Zaccone and Michael Davies. Dynamic web development with seaside. Square Bracket Associates, 2010.

[D'Ambros 2010] Marco D'Ambros, Michele Lanza and Romain Robbes. *Commit 2.0*. In Proceedings of the 1st Workshop on Web 2.0 for Software Engineering, Web2SE'10, pages 14–19. ACM, 2010.

[Ebraert 2007] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen and Theo D'Hondt. *Change-oriented software engineering*. In Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference, ICDL '07, pages 3–24. ACM, 2007.

[Ebraert 2008] Peter Ebraert. *First-Class Change Objects for Feature-Oriented Programming*. In Proceedings of the 15th Working Conference on Reverse Engineering, WCRE'08, pages 319–322. IEEE Computer Society, 2008.

[Ebraert 2010] Peter Ebraert, Theo D'Hondt, Tim Molderez and Dirk Janssens. *Intensional changes: modularizing crosscutting features*. In Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, pages 2176–2182. ACM, 2010.

[Ebraert 2011] Peter Ebraert, Quinten David Soetens and Dirk Janssens. *Change-based FODA diagrams: bridging the gap between feature-oriented design and implementation*. In Proceedings of the 2011 ACM Symposium on Applied Computing, SAC'11, pages 1345–1352. ACM, 2011.

[Edwards 1997] W. Keith Edwards. *Flexible conflict detection and management in collaborative applications*. In Proceedings of the 10th annual ACM symposium on User interface software and technology, UIST'97, pages 139–148, 1997. ACM.

[Elbaum 2000] Sebastian G. Elbaum, Alexey G. Malishevsky and Gregg Rothermel. *Prioritizing test cases for regression testing*. In Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'00, pages 102–112. ACM Press, 2000.

[Elbaum 2003] S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel and S. Kanduri. *Understanding the effects of changes on the cost-effectiveness of regression testing techniques*. Journal of Software Testing, Verification, and Reliability, vol. 13, no. 2, pages 65–83, June 2003.

[Fabry 2011] Johan Fabry, Andy Kellens, Simon Denier and Stéphane Ducasse. *AspectMaps: A Scalable Visualization of Join Point Shadows*. In Proceedings of the 19th International Conference on Program Comprehension, ICPC'11, pages 121–130. IEEE Computer Society Press, 2011.

[Feather 1989] Martin S. Feather. *Detecting interference when merging specification evolutions*. In IWSSD '89: Proceedings of the 5th international workshop on Software specification and design, pages 169–176, 1989. ACM.

[Fluri 2006] Beat Fluri and Harald C. Gall. *Classifying Change Types for Qualifying Change Couplings*. In Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC'06, pages 35–45, 2006. IEEE Computer Society.

[Fluri 2007] Beat Fluri, Michael Wuersch, Martin PInzger and Harald Gall. *Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction*. IEEE Transactions on Software Engineering, vol. 33, pages 725–743, 2007.

[Fritz 2010] Thomas Fritz and Gail C. Murphy. *Using information fragments to answer the questions developers ask*. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE'10, pages 175–184. ACM, 2010.

[Gallagher 1991] Keith Brian Gallagher and James R. Lyle. *Using Program Slicing in Software Maintenance*. Transactions on Software Engineering, vol. 17, no. 18, pages 751–761, August 1991.

[German 2009] Daniel M. German, Ahmed E. Hassan and Gregorio Robles. *Change impact graphs: Determining the impact of prior code changes*. Journal of Information Software Technology, vol. 51, no. 10, pages 1394–1408, October 2009.

[Gîrba 2005a] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Bern, Bern, November 2005.

[Gîrba 2005b] 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, pages 2–11, 2005. IEEE Computer Society.

[Gîrba 2006] Tudor Gîrba and Stéphane Ducasse. *Modeling History to Analyze Software Evolution*. Journal of Software Maintenance: Research and Practice (JSME), vol. 18, pages 207–236, 2006.

[Git 2005] Git. *Git: The fast version control system*. http://git-scm.com, 2005.

[Goldberg 1989] Adele Goldberg and Dave Robson. Smalltalk-80: The language. Addison Wesley, 1989.

[Grass 1992] J.E. Grass. *Object-Oriented Design Archeology with CIA++*. Computing Systems, vol. 5, no. 1, pages 5–67, 1992.

[Gulla 1991] Bjørn Gulla, Even-André Karlsson and Dashing Yeh. *Change-oriented version descriptions in EPOS*. Software Engineering Journal, vol. 6, no. 6, pages 378–386, November 1991.

[Hajiyev 2006] Elnar Hajiyev, Mathieu Verbaere and Oege de Moor. *CodeQuest: Scalable Source Code Queries with Datalog*. In Proceedings of the 20th European Conference on Object-Oriented Programming, ECOOP'06, pages 2–28. Springer-Verlag, 2006.

[Han 1997] Jun Han. *Supporting Impact Analysis and Change Propagation in Software Engineering Environments*. In Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (including CASE '97), STEP'97, page 172, 1997. IEEE Computer Society.

[Hassan 2004] Ahmed Hassan and Richard Holt. *Using Development History Sticky Notes to Understand Software Architecture*. In Proceedings of the 12th International Workshop on Program Comprehension, IWPC'04, pages 183–193. IEEE Computer Society, 2004.

[Hassan 2009] A. Hassan. *Predicting Faults Using the Complexity of Code Changes*. In Proceedings of the 31st International Conference on Software Engineering, ICSE'09, pages 78–88. IEEE Computer Society, 2009.

[Hattori 2009a] L. Hattori and M. Lanza. *An environment for synchronous software development*. In ICSE Companion, pages 223–226. IEEE, 2009.

[Hattori 2009b] L. Hattori and M. Lanza. *Mining the history of synchronous changes to refine code ownership*. In Proceedings of the 6th International Workshop on Mining Software Repositories, MSR'09, pages 141–150. IEEE, 2009.

[Hattori 2010] L. Hattori and M. Lanza. *Syde: a tool for collaborative software development*. In ICSE Tool demo, pages 235–238. ACM, 2010.

[Herzig 2010] Kim Sebastian Herzig. *Capturing the long-term impact of changes*. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE'10, pages 393–396. ACM, 2010.

[Herzig 2011] Kim Herzig and Andreas Zeller. *Mining Cause-Effect-Chains from Version Histories*. In Proceedings of the 22nd International Symposium on Software Reliability Engineering, ISSRE'11, pages 60–69. IEEE, 2011.

[Hindle 2005] Abram Hindle and Daniel German. *SCQL: A formal model and a query language for source control repositories*. In Proceedings of the 2nd International Workshop on Mining Software Repositories, MSR'05, pages 100–105, 2005.

[Hondt 1998] Koen De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Vrije Universiteit Brussel, Belgium, December 1998.

[Horwitz 1989] Susan Horwitz, Jan Prins and Thomas Reps. *Integrating Non-Interfering Versions of Programs*. ACM Trans. Program. Lang. Syst., vol. 11, no. 3, pages 345–387, 1989.

[Hou 2006] D. Hou and J. Hoover. *Using SCL to Specify and Check Design Intent in Source Code*. IEEE Transactions on Software Engineering, vol. 32, pages 404–423, 2006.

[Hunt 1976] James W. Hunt and M. Douglas McIlroy. *An Algorithm for Differential File Comparison*. Rapport technique 41, AT&T Bell Laboratories Inc, 1976.

[Hunt 1977] James W. Hunt and Thomas G. Szymanski. *A Fast Algorithm for Computing Longest Common Subsequences*. Commun. ACM, vol. 20, no. 5, pages 350–353, 1977.

[Jackson 1994] Daniel Jackson and David A. Ladd. *Semantic Diff: A Tool for Summarizing the Effects of Modifications*. In Proceedings of the International Conference on Software Maintenance, ICSM'94, pages 243–252. IEEE Computer Society, 1994.

[Janzen 2003] Doug Janzen and Kris de Volder. *Navigating and Querying Code Without Getting Lost*. In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD'03, pages 178–187, 2003. ACM.

[Kellens 2007] Andy Kellens, Kim Mens and Paolo Tonella. *A Survey of Automated Code-Level Aspect Mining Techniques*. Transactions on Aspect-Oriented Software Development, vol. 4, no. 4640, pages 143–162, 2007.

[Kellens 2011] Andy Kellens, Coen De Roover, Carlos Noguera, Reinout Stevens and Viviane Jonckers. *Reasoning over the Evolution of Source Code Using Quantified Regular Path Expressions*. In Proceedings of the 18th Working Conference on Reverse Engineering, WCRE'11, pages 389–393. IEEE Computer Society, 2011.

[Khanna 2007] Sanjeev Khanna, Keshav Kunal and Benjamin C. Pierce. *A formal investigation of Diff3*. In Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science, FSTTCS'07, pages 485–496. Springer-Verlag, 2007.

[Klint 2011] Paul Klint, Tijs van der Storm and Jurgen Vinju. EASY meta-programming with rascal, volume 6491 of *Lecture Notes in Computer Science*, Generative and Transformational Techniques in Software Engineering III, pages 222–289. Springer-Verlag, 2011.

[Kung 1994] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Y. Toyoshima and C. Chen. *Change impact identification in object oriented software maintenance*. In Proceedings of the International Conference on Software Maintenance, pages 202–211, 1994.

[Lanza 2001] Michele Lanza. *The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques*. In Proceedings of the International Workshop on Principles of Software Evolution, IWPSE'01, pages 37–42, 2001.

[Lanza 2004] Michele Lanza. *CodeCrawler — Polymetric Views in Action*. In Proceedings of the 19th IEEE International Conference on Automated Software Engineering, ASE'04, pages 394–395. IEEE CS Press, 2004.

[LaToza 2010] Thomas D. LaToza and Brad A. Myers. *Hard-to-answer questions about code*. In Evaluation and Usability of Programming Languages and Tools, PLATEAU 10, pages 8:1–8:6. ACM, 2010.

[Laval 2009] Jannik Laval, Simon Denier, Stéphane Ducasse and Andy Kellens. *Supporting Incremental Changes in Large Models*. In Proceedings of ESUG International Workshop on Smalltalk Technologies, IWST'09, pages 1–7, 2009.

[Laval 2011] Jannik Laval, Simon Denier, Stéphane Ducasse and Jean-Rémy Falleri. *Supporting Simultaneous Versions for Software Evolution Assessment*. Journal of Science of Computer Programming (SCP), vol. 76, no. 12, pages 1177–1193, May 2011.

[Law 2003] James Law and Gregg Rothermel. *Whole Program Path-Based Dynamic Impact Analysis*. In Proceedings of the 25th International Conference on Software Engineering, ICSE'03, pages 308–318. IEEE Computer Society, 2003.

[Leblang 1984] David B. Leblang and Robert P. Chase. *Computer-Aided Software Engineering in a distributed workstation environment*. SIGSOFT Software Engineering Notes, vol. 9, no. 3, pages 104–112, April 1984.

[Lethbridge 2004] Timothy Lethbridge, Sander Tichelaar and Erhard Plödereder. *The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering*. In Electronic Notes in Theoretical Computer Science, volume 94, pages 7–18, 2004.

[Lie 1989] A. Lie, R. Conradi, T. M. Didriksen and E.-A. Karlsson. *Change oriented versioning in a software engineering database*. In Proceedings of the 2nd International Workshop on Software configuration management, pages 56–65, 1989. ACM.

[Lienhard 2007] Adrian Lienhard, Adrian Kuhn and Orla Greevy. *Rapid Prototyping of Visualizations using Mondrian*. In Proceedings IEEE International Workshop on Visualizing Software for Understanding, Vissoft'07, pages 67–70, June 2007. IEEE Computer Society.

[Lindhom 2001] Tancred Lindhom. A 3-way merging algorithm for synchronizing ordered trees - the 3DM merging and differencing tool for XML. Master's thesis, Helsinki University of Technology, 2001.

[Lippe 1992] Ernst Lippe and Norbert van Oosterom. *Operation-based merging*. In Proceedings of the 5th ACM SIGSOFT symposium on Software Development Environments, SDE'92, pages 78–87, 1992. ACM Press.

[Livshits 2005] Benjamin Livshits and Thomas Zimmermann. *DynaMine: finding common error patterns by mining software revision histories*. SIGSOFT Software Engineering Notes, vol. 30, no. 5, pages 296–305, September 2005.

[Lubkin 1991] David Lubkin. *Heterogeneous configuration management with DSEE*. In Proceedings of the 3rd International Workshop on Software Configuration Management, pages 153–160. ACM, 1991.

[Lucas 1997] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Programming Technology Lab, Vrije Universiteit Brussel, Brussels, Belgium, 1997.

[Lungu 2010] Mircea Lungu, Michele Lanza, Tudor Gîrba and Romain Robbes. *The Small Project Observatory: Visualizing Software Ecosystems*. Science of Computer Programming, Elsevier, vol. 75, no. 4, pages 264–275, April 2010.

[Maes 1987] Pattie Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Belgium, January 1987.

[Marin 2007] Marius Marin, Arie van Deursen and Leon Moonen. *Identifying crosscutting concerns using fan-in analysis*. ACM Transactions on Software Engineering and Methodology, vol. 17, no. 1, pages 1–37, 2007.

[Marjanovic 2006] Dane Marjanovic. *Developing a Meta Model for Release History Systems*, 2006.

[Matthijssen 2010] Nick Matthijssen, Andy Zaidman, Margaret-Anne Storey, Ian Bull and Arie van Deursen. *Connecting Traces: Understanding Client-Server Interactions in Ajax Applications*. In Proceedings of the 18th IEEE International Conference on Program Comprehension, ICPC'10, pages 216–225. IEEE, 2010.

[Mens 1999] Tom Mens. *A formal foundation for object-oriented software evolution*. PhD thesis, Vrije Universiteit Brussel, September 1999.

[Mens 2000] Tom Mens. *Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution*. In Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance, AGTIVE'99, pages 127–143. Springer-Verlag, 2000.

[Mens 2002] T. Mens. *A State-of-the-Art Survey on Software Merging*. IEEE Transactions on Software Engineering, vol. 28, no. 5, pages 449–462, 2002.

[Mens 2004] Tom Mens and Tom Tourwé. *A Survey of Software Refactoring*. IEEE Transaction on Software Engineering, vol. 30, no. 2, pages 126–139, 2004.

[Meyer 2006] Michael Meyer, Tudor Gîrba and Mircea Lungu. *Mondrian: An Agile Visualization Framework*. In ACM Symposium on Software Visualization, SoftVis'06, pages 135–144, 2006. ACM Press.

[Meyers 2010] Bart Meyers, Peter Ebraert and Dirk Janssens. *Intensional changes avoid co-evolution!* In Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE'10, pages 4:1–4:6. ACM, 2010.

[Mikhajlov 1998] Leonid Mikhajlov and Emil Sekerinski. *A Study of the Fragile Base Class Problem*. In Proceedings of the European Conference on Object-Oriented Programming, numéro 1445 de Lecture Notes in Computer Science, pages 355–383. Springer-Verlag, 1998.

[Necula 2002] George C. Necula, Scott McPeak, Shree Prakash Rahul and Westley Weimer. *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*. In International Conference on Compiler Construction, numéro 2304 de Lecture Notes in Computer Science, pages 213–228, April 2002.

[Nielsen 1993] Jakob Nielsen. Usability engineering. Morgan Kaufmann, 1st édition, September 1993.

[Nierstrasz 2005] Oscar Nierstrasz, Stéphane Ducasse and Tudor Gîrba. *The Story of Moose: an Agile Reengineering Environment*. In Michel Wermelinger and Harald Gall, editeurs, Proceedings of the European Software Engineering Conference, ESEC/FSE'05, pages 1–10, 2005. ACM Press. Invited paper.

[Padioleau 2008] Y. Padioleau, J. Lawall and G. Muller. *Documenting and automating collateral evolutions in linux device drivers*. In Proceedings of the 3rd SIGOPS/EuroSys European Conference on Computer Systems, EuroSys'08, pages 247–260. ACM, 2008.

[Pelrine 2001] Joseph Pelrine and Alan Knight. Mastering envy/developer. Cambridge University Press, 2001.

[Perry 2001] Dewayne E. Perry, Harvey P. Siy and Lawrence G. Votta. *Parallel changes in large-scale software development: an observational case study*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 10, no. 3, pages 308–337, July 2001.

[Pfeiffer 2006] J.-Hendrik Pfeiffer and John R. Gurd. *Visualisation-based tool support for the development of aspect-oriented programs*. In Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD'06, pages 146–157. ACM, 2006.

[Phillips 2011] Shaun Phillips, Jonathan Sillito and Rob Walker. *Branching and merging: an investigation into current version control practices*. In Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE'11, pages 9–15. ACM, 2011.

[Phillips 2012] Shaun Phillips, Guenther Ruhe and Jonathan Sillito. *Information needs for integration decisions in the release process of large-scale parallel development*. In Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work, CSCW'12, pages 1371–1380. ACM, 2012.

[Premraj 2011] Rahul Premraj, Antony Tang, Nico Linssen, Hub Geraats and Hans van Vliet. *To branch or not to branch?* In Proceedings of the 2011 International Conference on Software and Systems Process, ICSSP'11, pages 81–90. ACM, 2011.

[Ren 2004] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder and Ophelia Chesley. *Chianti: A tool for change impact analysis of Java programs*. In Proceedings of the Object-Oriented Programming, Systems, Languages & Applications, OOPSLA'04, pages 432–448, oct 2004. ACM.

[Ren 2006] Xiaoxia Ren, Ophelia C. Chesley and Barbara G. Ryder. *Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis.* IEEE Transactions on Software Engineering, vol. 32, no. 9, pages 718–732, September 2006.

[Rho 1998] Jungkyu Rho and Chisu Wu. *An Efficient Version Model of Software Diagrams.* In Proceedings of the 5th Asia Pacific Software Engineering Conference, APSEC'98, pages 236–243, 1998. IEEE Computer Society.

[Rivard 1996] Fred Rivard. *Reflective Facilities in Smalltalk.* Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996, February 1996.

[Robbes 2005] Romain Robbes, Stéphane Ducasse and Michele Lanza. *Microprints: A Pixel-based Semantically Rich Visualization of Methods.* In Proceedings of 13th International Smalltalk Conference, ISC'05, pages 131–157, 2005.

[Robbes 2007] Romain Robbes and Michele Lanza. *A Change-based Approach to Software Evolution.* Electronic Notes in Theoretical Computer Science, vol. 166, pages 93–109, January 2007.

[Robbes 2008a] Romain Robbes. *Of Change and Software.* PhD thesis, University of Lugano, Switzerland, December 2008.

[Robbes 2008b] Romain Robbes and Michele Lanza. *SpyWare: a change-aware development toolset.* In Proceedings of the 30th International Conference on Software Engineering, ICSE'08, pages 847–850, 2008. ACM.

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

[Roberts 1999] Donald Bradley Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois, 1999.

[Rochkind 1975] Marc Rochkind. *The Source Code Control System.* IEEE Transactions on Software Engineering, vol. 1, no. 4, pages 364–370, 1975.

[Ryder 2001] Barbara G. Ryder and Frank Tip. *Change impact analysis for object-oriented programs.* In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 46–53. ACM Press, 2001.

[Schärli 2003] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew P. Black. *Traits: Composable Units of Behavior.* In Proceedings of European Conference on Object-Oriented Programming, volume 2743 of *ECOOP'03*, pages 248–274. Springer Verlag, July 2003.

[Shao 2007] D. Shao, S. Khurshid and D.E. Perry. *Evaluation of semantic interference detection in parallel changes: an exploratory experiment.* In Proceedings of the 23rd IEEE International Conference on Software Maintenance, ICSM'07, pages 74–83, oct 2007.

[Shao 2009] Danhua Shao, Sarfraz Khurshid and Dewayne E. Perry. *SCA: a Semantic Conflict Analyzer for Parallel Changes.* In Proceedings of the the 7th joint meeting of the European

Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering, ESEC/FSE'09, pages 291–292, 2009.

[Shen 2004] Haifeng Shen and Chengzheng Sun. *A Complete Textual Merging Algorithm for Software Configuration Management Systems*. In Proceedings of the 28th Annual International Computer Software and Applications Conference, COMPSAC'04, pages 293–298, 2004. IEEE Computer Society.

[Sillito 2005] Jonathan Sillito, Kris De Volder, Brian Fisher and Gail Murphy. *Managing software change tasks: An exploratory study*. In Proceedings of the International Symposium on Empirical Software Engineering, pages 23–32. IEEE Computer Society, 2005.

[Sillito 2006] J. Sillito, G.C. Murphy and K. De Volder. *Questions Programmers Ask During Software Evolution Tasks*. In Proceedings of the 14th International Symposium on Foundations on Software Engineering, SIGSOFT '06/FSE-14, pages 23–34. ACM, 2006.

[Sillito 2008] J. Sillito, G.C. Murphy and K. De Volder. *Asking and Answering Questions during a Programming Change Task*. IEEE Transactions on Software Engineering, vol. 34, no. 4, pages 434–451, jul 2008.

[Stasko 1998] John T. Stasko, John Domingue, Marc H. Brown and Blaine A. Price. Software visualization — programming as a multimedia experience. The MIT Press, 1998.

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

[Stoerzer 2006] Maximilian Stoerzer, Barbara G. Ryder, Xiaoxia Ren and Frank Tip. *Finding Failure-Inducing Changes in Java Programs using Change Classification*. In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pages 57–68. ACM, nov 2006.

[Storey 1997a] Margaret-Anne D. Storey, Kenny Wong, F. D. Fracchia and Hausi A. Müller. *On integrating visualization techniques for effective software exploration*. In Proceedings of IEEE Symposium on Information Visualization, InfoVis'97, pages 38–48. IEEE Computer Society, 1997.

[Storey 1997b] Margaret-Anne D. Storey, Kenny Wong and Hausi A. Müller. *How Do Program Understanding Tools Affect How Programmers Understand Programs?* In Proceedings of the 4th Working Conference on Reverse Engineering, pages 12–21. IEEE Computer Society, 1997.

[Taenzer 1989] David Taenzer, Murthy Ganti and Sunil Podar. *Problems in Object-Oriented Software Reuse*. In Proceedings of the 3rd European Conference on Object-Oriented Programming, ECOOP'89, pages 25–38, 1989.

[Takang 1996] Armstrong A. Takang, Penny A. Grubb and Robert D. Macredie. *The effects of comments and identifier names on program comprehensibility: an experimental investigation.* Journal of Programming Languages, vol. 4, no. 3, pages 143–167, 1996.

[Thione 2005] Gian Lorenzo Thione and Dewayne E. Perry. *Parallel Changes: Detecting Semantic Interferences.* In Proceedings of the 29th International Computer Software and Applications Conference, COMPSAC'05, pages 47–56. IEEE Computer Society, 2005.

[Thomas 1988] Dave Thomas and Kent Johnson. *Orwell — A Configuration Management System for Team Programming.* In Proceedings of the Object-Oriented Programming, Systems, Languages & Applications, ACM SIGPLAN Notices, volume 23 of *OOPSLA'88*, pages 135–141, November 1988.

[Tichelaar 2000] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer and Oscar Nierstrasz. *A Meta-model for Language-Independent Refactoring.* In Proceedings of International Symposium on Principles of Software Evolution, ISPSE'00, pages 157–167. IEEE Computer Society Press, 2000.

[Tichy 1982] Walter F. Tichy. *Design, implementation, and evaluation of a Revision Control System.* In Proceedings of the 6th International Conference on Software Engineering, ICSE'82, pages 58–67. IEEE Computer Society Press, 1982.

[Tichy 1984] Walter F. Tichy. *The string-to-string correction problem with block moves.* Journal of ACM Transactions on Computer Systems (TOCS), vol. 2, no. 4, pages 309–321, November 1984.

[Tichy 1985] Walter F. Tichy. *RCS—a system for version control.* Software Practice and Experience, vol. 15, no. 7, pages 637–654, July 1985.

[Tip 1995] Frank Tip. *A survey of program slicing techniques.* Journal of Programming Languages, vol. 3, pages 121–189, 1995.

[Tufte 2001] Edward R. Tufte. The visual display of quantitative information. Graphics Press, 2nd édition, 2001.

[Uquillas-Gómez 2009] Verónica Uquillas-Gómez, Andy Kellens, Johan Brichau and Theo D'Hondt. *Time warp, an approach for reasoning over system histories.* In Proceedings of the joint International and annual ERCIM workshops on Principles of Software Evolution, and Software Evolution workshops, IWPSE-Evol'09, pages 79–88. ACM, 2009.

[Uquillas Gómez 2010a] Verónica Uquillas Gómez, Stéphane Ducasse and Theo D'Hondt. *Meta-models and Infrastructure for Smalltalk Omnipresent History.* In Smalltalks'2010, 2010.

[Uquillas Gómez 2010b] Verónica Uquillas Gómez, Stéphane Ducasse and Theo D'Hondt. *Visually Supporting Source Code Changes Integration: the Torch Dashboard.* In Proceedings of the 17th Working Conference on Reverse Engineering, WCRE'10, pages 55–64, October 2010.

[Uquillas Gómez 2012] Verónica Uquillas Gómez, Stéphane Ducasse and Theo D'Hondt. *Ring: a Unifying Meta-Model and Infrastructure for Smalltalk Source Code Analysis Tools.* Computer Languages, Systems and Structures, vol. 38, no. 1, pages 44–60, April 2012.

[van den Hamer 1996] Peter van den Hamer and Kees Lepoeter. *Managing Design Data: The Five Dimensions of CAD Frameworks, Configuration Management, and Product Data Management*. In Proceedings of the IEEE, volume 84, pages 42 – 56. IEEE CS Press, January 1996.

[Van der Heijden 1996] A. H. C. Van der Heijden. *Perception for selection, selection for action, and action for perception*. Visual Cognition, vol. 3, no. 4, pages 357–361, December 1996.

[Vogel 2012] Lars Vogel. *Eclipse Modeling Framework (EMF) - Tutorial*. http://www.vogella.com/articles/EclipseEMF/article.html, 2012.

[Walrad 2002] C. Walrad and D. Strom. *The importance of branching models in SCM*. Computer, vol. 35, no. 9, pages 31–38, 2002.

[Ware 2004] Colin Ware. Information visualisation. Elsevier, Sansome Street, San Fransico, 2004.

[Weiser 1981] Mark Weiser. *Program slicing*. In Proceedings of the 5th International Conference on Software Engineering, ICSE'81, pages 439–449, 1981. IEEE Press.

[Welser 1984] M. Welser. *Program Slicing*. IEEE Transactions on Software Engineering, pages 352–357, 1984.

[Westfechtel 1991] Bernhard Westfechtel. *Structure-oriented merging of revisions of software documents*. In Proceedings of the 3rd international workshop on Software configuration management, pages 68–79, 1991. ACM.

[Wilde 1992] Norman Wilde and Ross Huitt. *Maintenance Support for Object-Oriented Programs*. IEEE Transactions on Software Engineering, vol. 18, no. 12, pages 1038–1044, December 1992.

[Wloka 2009] Jan Wloka, Barbara Ryder, Frank Tip and Xiaoxia Ren. *Safe-commit analysis to facilitate team software development*. In Proceedings of the 31st International Conference on Software Engineering, ICSE'09, pages 507–517. IEEE Computer Society, 2009.

[Wuyts 2001] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[Yang 1991] Wuu Yang. *Identifying syntactic differences between two programs*. Software Practice & Experience, vol. 21, no. 7, pages 739–755, June 1991.

[Yang 1992] Wuu Yang, Susan Horwitz and Thomas Reps. *A program integration algorithm that accommodates semantics-preserving transformations*. ACM Transactions on Software Engineering and Methodology, vol. 1, no. 3, pages 310–354, July 1992.

[Yang 1994] Wuu Yang. *How to merge program texts*. Journal of Systems and Software, vol. 27, no. 2, pages 129–135, November 1994.

[Zhang 2008] Sai Zhang, Zhongxian Gu, Yu Lin and Jianjun Zhao. *Change Impact Analysis for AspectJ Programs*. In Proceedings of the 24th IEEE International Conference on Software Maintenance, ICSM'08, pages 87–96. IEEE, 2008.

[Zimmermann 2004a] Thomas Zimmermann and Peter Weißgerber. *Preprocessing CVS Data for Fine-Grained Analysis*. In Proceedings of the 1st International Workshop on Mining Software Repositories, MSR'04, pages 2–6, 2004. IEEE Computer Society Press.

[Zimmermann 2004b] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl and Andreas Zeller. *Mining Version Histories to Guide Software Changes*. In Proceedings of the 26th International Conference on Software Engineering, ICSE'04, pages 563–572. IEEE Computer Society Press, 2004.