Visually Characterizing Source Code Changes

Verónica Uquillas-Gomez, Stéphane Ducasse, Theo d'Hondt

Visually Characterizing Source Code Changes

Verónica Uquillas Gómez[a,b], Stéphane Ducasse[b], Theo D'Hondt[a]

[a] *Software Languages Lab — Vrije Universiteit Brussel — Belgium*
[b] *RMoD Team — INRIA Lille-Nord Europe Research Center*
Laboratoire d'Informatique Fondamentale de Lille — Université Lille1 — France

Abstract

Revision Control Systems (*e.g.,* SVN, Git, Mercurial) include automatic and advanced merging algorithms that help developers to merge their modifications with development repositories. While these systems can help to textually detect conflicts, they do not help to identify the semantic consequences of a change. Unfortunately, there is little support to help *release masters* (integrators) to take decisions about the integration of changes into the system release. Most of the time, the release master needs to read all the modified code, check the diffs to build an idea of a change, and dig for details from related unchanged code to understand the context and potential impact of some changes. As a result, such a task can be overwhelming. In this article we present a visualization tool to support integrators of object-oriented programs in comprehending changes. Our approach named Torch characterizes changes based on structural information, authors and symbolic information. It mixes text-based diff information with visual representation and metrics characterizing the changes. The current implementation of our approach analyses Smalltalk programs, and thus we describe our experiments applying it to Pharo, a large open-source system. We also report on the evaluations of our approach by release masters and developers of several open-source projects.

Keywords: Source code, visualizations, change understanding

This article makes heavy use of colors. Please read a color print of this article to better understand the presented ideas.

1. Supporting Change Integration

Integrating changes that represent fixes, enhancements or new features are key software development activities. However, integrating these changes in a large-scale collaborative software development environment poses substantial

Email addresses: vuquilla@vub.ac.be (Verónica Uquillas Gómez), stephane.ducasse@inria.fr (Stéphane Ducasse), tjdhondt@vub.ac.be (Theo D'Hondt)

challenges. 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.

Merging techniques used by popular revision control systems (*e.g.,* CVS, Subversion, Git) are based on simple, text-based algorithms, and are therefore oblivious to the program entities they merge. These systems can help to textually detect conflicts [3], but they do not help to identify semantic consequences of a change. Even though there exist other approaches providing advanced merging support([1, 30]) 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.

There is no adequate support to help integrators to take decisions about the integration of changes into the system release or internal development branches. We identified two main problems: First, integrators do not get an overview of the changes: how changes are distributed over the system, what groups of entities changed. For example, large but regular changes are often easier to integrate than smaller but more complex ones. Second, integrators do not get the possibility to understand detailed changes within their specific context.

In this article, we focus on developers that support the production of a new version. From this perspective and ignoring issues related to testing (acceptation testing and others), we can identify two roles: *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*** analyse the code of *committers*, merge selected changes (which were made to previous versions) and release them into the current version.

There are several merging techniques: text-based [41, 3, 25], syntactic-based [5, 2, 44], semantic-based [44, 4], operation-based [35, 9] and merging algorithms such as two-way merge [22] and three-way merge [26]. Several tools such as Envy [40] take into account the underlying meta-model as a step towards a semantic merge. Still, 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 a clear lack of support. In this article, we present *Torch*, a dashboard displaying object-oriented[1] changes using several visualizations and change summaries that support integrators and developers in comprehending changes. Especially, Torch provides

[1]The current version of Torch analyses systems implemented in Smalltalk but using a language independent code model.

four visualizations based on package distribution and class hierarchies while at the same time it offers *an omnipresent contextual diff based on a fly-by help*[2].

This article is an extension of our WCRE 2010 paper [42] where the original contributions were: (1) identification of integrator needs and (2) a change dashboard, named *Torch* that aims at providing a means to understand changes.

Our new contributions in this article are: (1) a description of the underlying infrastructure and models, and (2) an extended evaluation that includes an experiment showing that developers are efficient with Torch. Moreover, we provide a new elaborated description of multiple example usages of Torch,

The article is structured as follows. In Section 2 we present the challenges of characterizing changes. Torch is introduced in Section 3, followed by a detailed definition of the different visualizations in Section 4. We illustrate the use of our approach in Section 5 showing six example usages. The infrastructure of Torch is explained in Section 6. In Section 7, we present a discussion of the evaluation of Torch. Section 8 is dedicated to discussing related work, and we conclude this article in Section 9.

2. Changes Characterization Challenges

Understanding changes is required in daily software development activities. Integrators and developers need to comprehend changes before actually merging these changes with the system release or internal development branches. For this, they rely on revision control systems to extract patches. Patches are changes of source code that do not track the complete history of actions that led to the changes [33, 17]. From that perspective, operation-based merge [27, 9] is ruled out since it is based on the idea that either refactorings or every single action made by the programmer is fully recorded.

The state of the art in industry and open-source development is often limited to good diff tools (*e.g.,* Guiffy in Eclipse or Monticello in Smalltalk). They show the changes as code snippets that 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.

For three years, the second author of this article was one of the integrators of the Squeak open-source project. Since four years now, he is one of the integrators of the Pharo open-source project. Squeak 4.3 is composed of 2105 classes distributed over 63 packages. Pharo 2.0 is composed of 3286 classes for 242 packages. Both systems evolved in an ecosystems composed of around 800 to 1000 public projects.

From this experience we assemble a non-exhaustive list of challenges which integrators and developers face daily, and a list of characteristics of changes needed to partially tackle such challenges. Note that our solution does not

[2] A fly-by help is a tooltip that appears when the user hovers the pointer over an item on the visualization.

provide a solution for the challenges presented here but it constitutes a reference for change characterization and change integration challenge identification.

Challenges:

- *Architectural drifting.* When integrating a change, it is difficult to assess whether a change is not breaking hidden assumptions about the architecture of a system. The introduction of code using inadequate classes may tie parts that should stay independent or go against established constraints (*e.g.,* when migrating from one framework to another).

- *Change and conflicts.* Often a committer performs a change against an old version of the system. Two questions then arise: what was the delta in the context of the version of the system at the time of the change and how should that delta be interpreted in the presence of the current trunk version.

- *Impact of the changes.* A much more difficult problem is to understand the impact of a change. This is particularly difficult in presence of "yoyo effect" [45, 39] and *fragile base class problem* [36]. The problem is that a simple change may break existing framework customizations. In such a context the location in the inheritance hierarchy is a first step to assess how many subclasses are impacted by a change and to determine their clients.

- *Test regression assessment.* An integrator is often under stress due to the fact that some changes should be integrated whereas at the same there is no guarantee that no new bugs get introduced. Assessing test regression is a key aspect especially in presence of complex code.

Characteristics:

- *Size.* Characterizing a change in terms of its size gives a first impression of a change. The first measure is the size of the changes in terms of lines of code impacted. Note that size is just indicative since a small change can have huge effects.

- *Structure.* The packages, classes and methods are the core of programs and can be used by the tools. The number of packages, classes and methods compared to the application size is another simple characterization of a change. Such an estimate can be misleading, for example when a simple API use change is performed throughout a complete system.

- *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.,* added or removed) or intra (*i.e.,* modified code) is another element. Whether the changes were actually really changing behavior (*e.g.,* not just changes to license or comments) is complementary to the other information.

- *Vocabulary.* In certain situations, assessing the difference in vocabulary between a change and the application can give information about whether or not that change fits the existing application.

- *Change scope.* Assessing a well-scoped change is often simpler than one crosscutting several hierarchies or packages. Therefore, getting a fast overview of the location of changed elements in the context of the hierarchy and package structure is important.

- *Dependent and correlated changes.* A specific change can require several other changes. This is especially the case when changes come from different branches or forks. Knowing that a piece of code has always been changed together with some other pieces of code can be key in spotting problems with a change.

3. Supporting Change Understanding with Torch

Overview. To support change integration, our approach characterizes changes according to structural, authorial and symbolic information. The integrator or developer can see contextual diffs representing the changes using class hierarchy or package distribution visualizations.

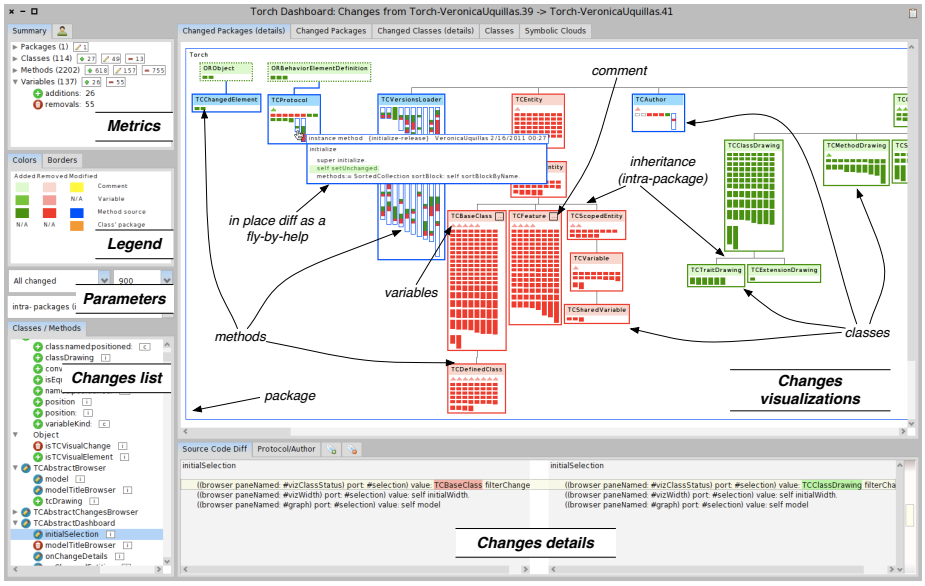

Figure 1: Dashboard main elements: the *metrics* give an idea of the size of the changed entities and the actual changes; the *changes list* presents the list of changes and their detailed difference using the *changes details*; the *changes visualizations* present a map of changes structured around packages and classes.

Torch [3] provides visual tool support to integrators and developers to characterize and comprehend changes in context. It helps them in taking decisions about the integration of changes before performing the actual merging. In addition, Torch also offers developers a means to understand and control their changes before publishing them.

Based on a literature survey we identified the information that can help to characterize changes and address some of the challenges previously identified.

The Torch dashboard (shown in Figure 1) contains a set of metrics about changes per program entity and per author. Several visualizations showing the structural representations of changes are at the core of the dashboard. It includes a contextual diff as a fly-by help on top of the visualizations to speed up the access to the textual information of changes. 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 users 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.

The dashboard provides information based on the Torch change-based model that represents changes between two versions and their context. This model is described in Section 6.2.

Figure 1 shows the structure and main elements of the Torch dashboard. In this section, we only present a short overview of each element. In Section 4 we present a more extended description of the *Changes visualizations* element.

Metrics. They 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 panel *Summary* shows information per program entity (*i.e.,* packages, classes, methods, variables) and per kind of action (*i.e.,* added, modified, removed). The panel *Authors* (not shown in the figure) 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. 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

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

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 by demand using the *relationships* parameter.

Changes list. Changes representing classes and methods are listed here. They are retrieved from the change-based model and can be filtered out by selecting metrics or entities from the visualizations.

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 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 1). The changes are highlighted respecting the *Conventions*. The visualizations include a contextual fly-by help that supports an in-place diff view.

Torch is developed in Smalltalk and is available for Pharo[4]. Torch does not use the Smalltalk code meta-model but its own change-based code representation built on top of the Ring source code meta-model [43]. Torch relies on the information provided by the Monticello SCM system and is integrated with the Monticello tools to give developers direct access to versions (or group of versions) in a repository. However, Torch can be adapted to other versioning systems like Git.

4. Dashboard Visualizations in Details

The comparison of two versions is graphically presented in the main element of the dashboard, named *Changes visualizations*. It shows program entities, their relationships, and the vocabulary involved in changes. Additionally, Torch does not only show changed entities but also unchanged ones, providing a complete visual structural representation of each version with the context and characteristics of changes.

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.

[4]Pharo: http://www.pharo-project.org

In object-oriented programs two main definitions are available for structuring a system: the packages and the class inheritance hierarchies. 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" [39, 45]. Even an enhanced 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.

4.1. Entities Representation

Torch uses two shapes for representing entities: rectangles and triangles. Rectangles represent packages, classes and methods; triangles represent attributes. Dashed borders are used for traits and extensions. Three kinds of edges are used for representing relationships (*i.e.*, class-inherits-class (arrowed edges), class-uses-trait (dashed arrowed edges) and class-is-extended-in-package (dashed edges)). Colors are mapped onto a type of change of an entity or inheritance relationship.

Packages. Figure 2 shows the modified System-FilePackage and its changed classes using structural representation of classes. A package is displayed as a large rectangle containing its classes (even unchanged ones). Inside, when possible, classes are organized in class hierarchies, and they show changes using any of the class representations explained later. Unchanged classes are represented by small dashed boxes.

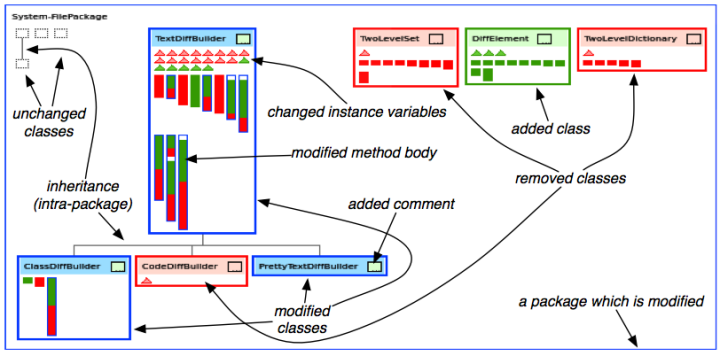

Figure 2: Package containing unchanged classes (small dashed grey rectangles), removed (red rectangles), added (green rectangle) and modified classes (blue rectangles). Classes contain attributes (triangles) and methods (bars).

Classes. A class has two visual representations for its changes: structural representation and condensed representation. Figure 3 shows both class representations[5] making use of three classes (added, removed and modified class). Note

[5]Due to space restrictions, our examples only use the structural representation of classes.

that the color of the border of a class and a light color as the background of the class name represent whether the class was entirely added (green), removed (red) or simply modified (blue). Our class representations may also display a colored box beside the class name for *changed class comment*.

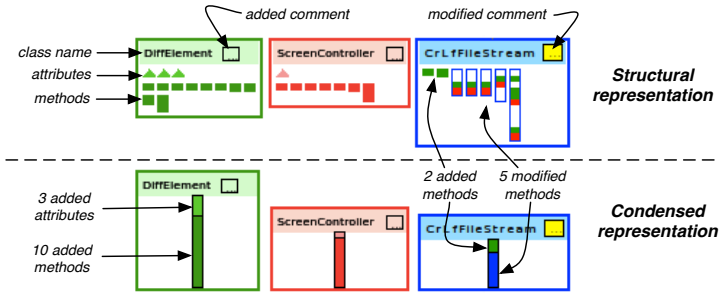

Figure 3: Structural and condensed visual representation of classes.

- *Structural representation.* A class is displayed using sections: class name section, attributes section and methods section (see classes on the top of Figure 3). DiffElement and ScreenController have changed attributes, methods and comment, whereas CrLfFileStream has changed methods and comment, therefore the attributes section is hidden. 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).

- *Condensed representation.* Changed attributes and methods may also be presented together as a single bar summarizing the amount of changes (see classes at the bottom of Figure 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.

4.2. Fly-by Help

Diff as a fly-by help. The main visualization of the dashboard shows the structural representation of changed classes 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 4 shows a source diff as a fly-by help. It shows the method's code and highlights line additions in red and removals in green. 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:

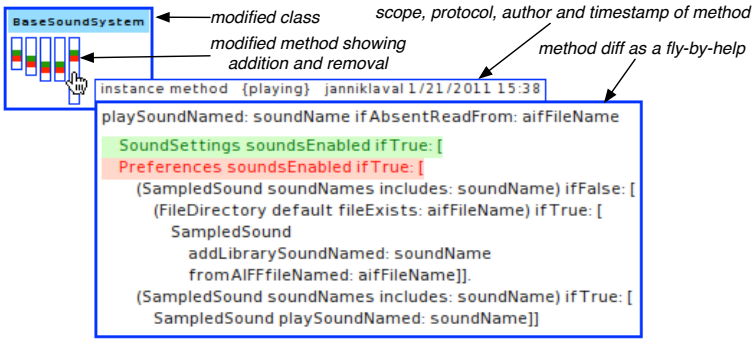

Figure 4: Omnipresent code browsing: diff as a fly-by help.

the scope (*i.e.,* instance or class method), the protocol[6], the author and the timestamp when the change happened.

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 (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 amount 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.

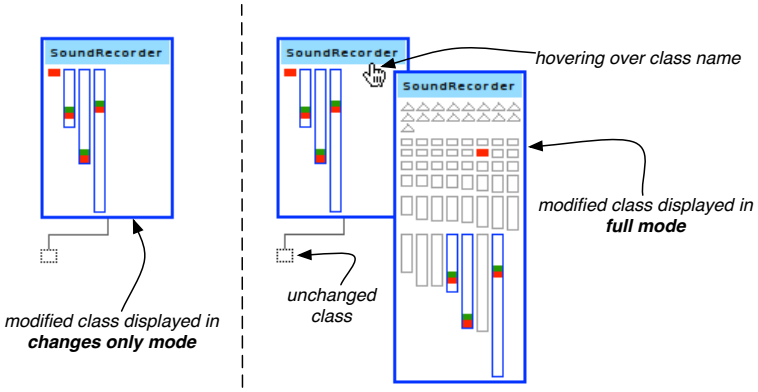

Figure 5: Class displayed in changes only mode on the left, and in the full mode as a fly-by help on the right.

[6]Methods in Smalltalk are classified in protocols.

4.3. Package-centric Visualizations

Package-centric visualizations provide the structural context of any existing change, by distributing classes in packages and methods in classes. 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 task.

- *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 only focus on what was changed in that class.

- *Changed Packages (condensed).* This visualization also presents only changed packages, but 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 or stream of versions. An integrator can compare the size of changed vs. unchanged packages and can observe and explore classes defined in unchanged packages that may have relationships with changed classes (*e.g.,* inheritance).

As mentioned earlier inheritance-based or class-centric visualizations are also proposed using the same principles. An example usage of this visualization is shown in Figure 10.

4.4. Symbolic Clouds

Symbolic Clouds are the third kind of visualization presented in the dashboard. They show the vocabulary involved in changes 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 of the one of the application or new vocabulary is introduced).

The clouds are built by extracting method invocations, class references, and access 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 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 6 shows the two first clouds applied to the scenario where

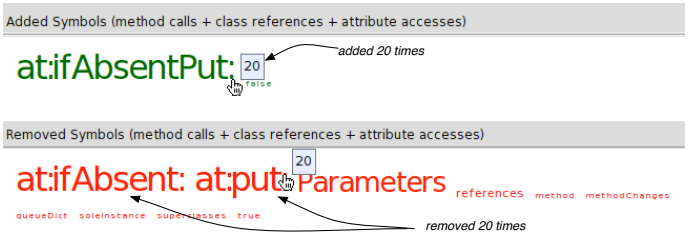

Figure 6: Added and removed symbolic clouds.

the combined method calls at:ifAbsent: and at:put: (red symbols) were replaced by the method call at:ifAbsentPut: (green symbol). Figure 12 shows another example usage based on the mixed symbolic cloud.

5. Torch Example Usages

In this section, we illustrate the usage of Torch by applying it to the change streams of the Pharo project. We took the repositories www.squeaksource.com/PharoInbox and ss3.gemstone.com/ss/PharoInbox containing the submissions of versions 1.3 and 1.4, and a third repository www.squeaksource.com/PharoTreatedInbox containing the submissions once they have been integrated into the current released version. 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 is applied to the following scenarios[7]:

- Removing a feature;
- Removing a feature and deprecating its API;
- Introducing a feature;
- Pushing up methods / Introducing methods in a class hierarchy;
- Adding comments;
- Replacing method calls.

5.1. Removing a feature

Replacing old features or cleaning dead code usually results in feature removals. A user can easily detect a feature removal with the Torch dashboard. The pattern is simple (*i.e.,* mostly removed entities which appear all in red) but it can be subtle: indeed clients may need to change not to refer to the removed features anymore. 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.

[7]The second and four scenario are reused from our previous work [42].

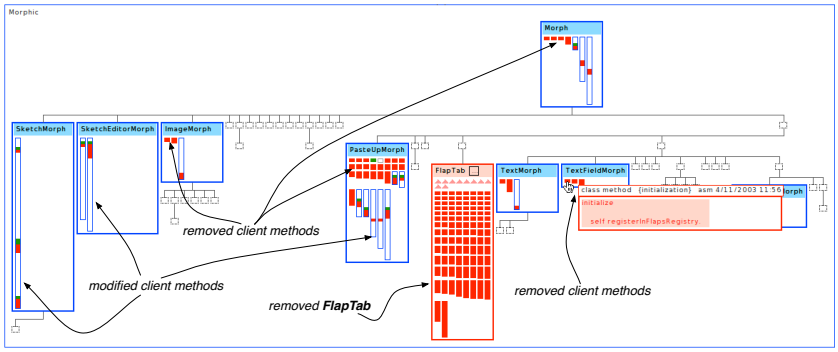

Figure 7: Removing the feature FlapTab: several methods in clients were *modified* and other methods were simply *removed*.

Figure 7 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).

5.2. Removing a feature and deprecating its API

Changes associated to a feature removal are mostly deletions of source code. However, the complete removal is often a practice that is not 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 Pharo there 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 8 shows the effect of this operation. 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 before and after the operation is shown in the 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. All other changes (*i.e.,* modifications in methods) correspond to the clients of PointerFinder that now make use of the new implementation.

5.3. 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.

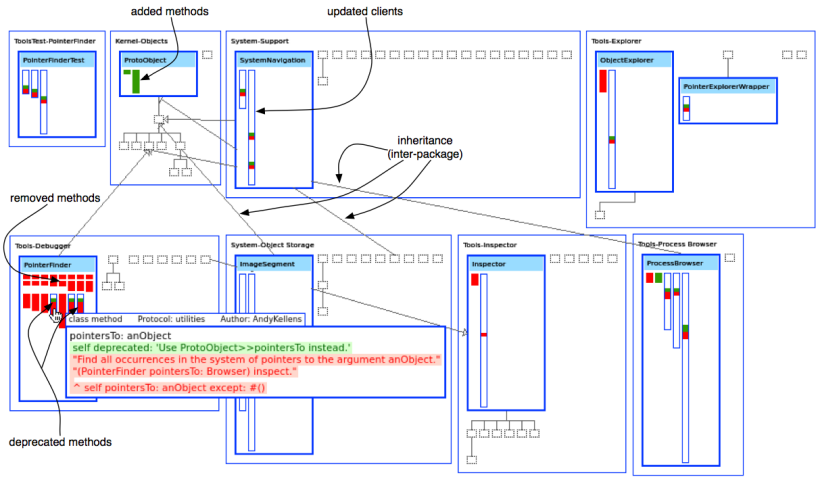

Figure 8: Removing the feature PointerFinder and deprecating its API: its functionality was substituted by another tool.

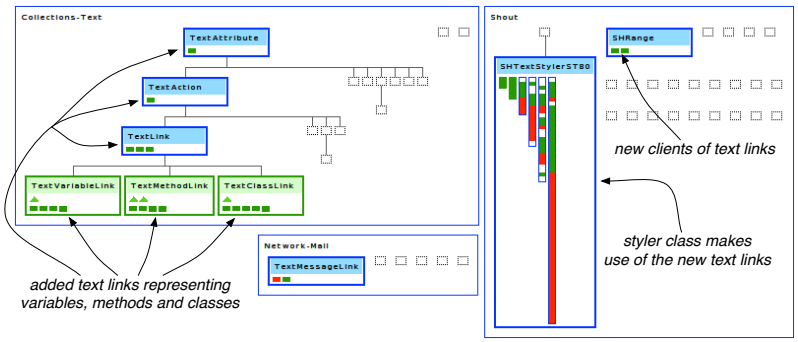

Figure 9: Introducing features: new variations of text links for code styler.

Figure 9 shows the introduction of three variations of text hyperlinks – TextClassLink, TextMethodLink and TextVariableLink – mainly used by the code styler feature SHTextStylerST80. The boxes of the three classes have 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 on 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 (both not displayed in the figure).

5.4. 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 gives immediate hints to integrators about the impact of changes within an inheritance tree (*i.e.,* class-centric visualization).

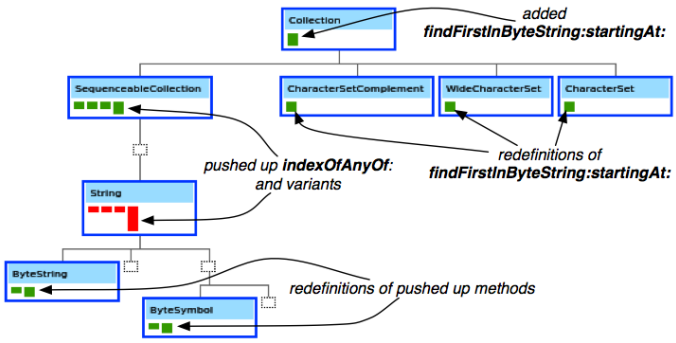

Figure 10: Pushing up methods in the SequenceableCollection class hierarchy, and introducing a method in the Collection class hierarchy.

Figure 10 shows the method indexOfAnyOf: and its variants -originally implemented in String (*i.e.,* removed methods)- pushed up to its indirect superclass SequenceableCollection and their redefinitions in two subclasses of String (*i.e.,* added methods). This example also shows the introduction of findFirstInByteString:startingAt: in Collection, the top superclass of this hierarchy, and its redefinitions (*i.e.,* added methods) in CharacterSetComplement, WideCharacterSet and CharacterSet. The classes of this scenario are defined in three packages, thus the class-centric visualizations provide a better view of the whole hierarchy. Looking at the changes using the class inheritance view can be more appropriate when package structure can be neglected.

5.5. Adding comments

Non-functional changes, such as additions or modifications of class/methods 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 will 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 will know that even though a change can be large in number of modified entities, the changes are linked to comments.

Figure 11 shows the addition and modification of comments in the graphical TickSelection classes, and a couple of comment additions at method level. Basically, in this case the developer documented these set of classes, which is displayed on the dashboard as the colored boxes next to the class names. Note that by providing this characterization of changes, a user can mainly focus on understanding the changed methods.

5.6. Replacing method calls

Introducing enhancements to a system may be represented by a set of modified methods that replace method calls (*i.e.,* changing old functionality for new

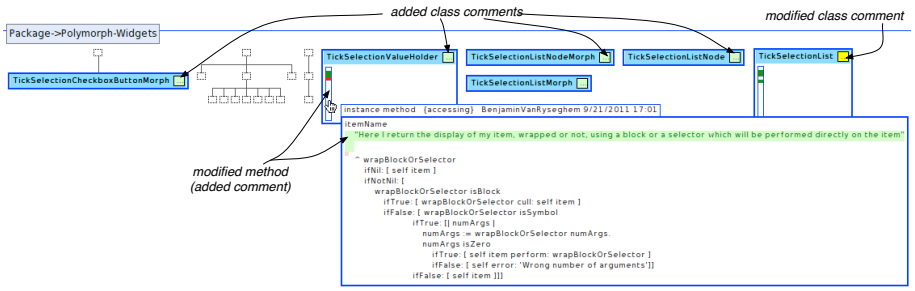

Figure 11: Adding comments: documenting the graphical TickSelection morph classes.

functionality). Depending on how many clients call those methods, a high number of changes may be produced. The integration of this kind of changes can also demand a lot of time from users as they will probably inspect every change. The symbolic clouds provided by Torch aim at showing the relevant vocabulary involved in a change, and for this scenario the symbolic clouds fit perfectly as they will show few symbols referring to the old an new calls, each with a high occurrence.

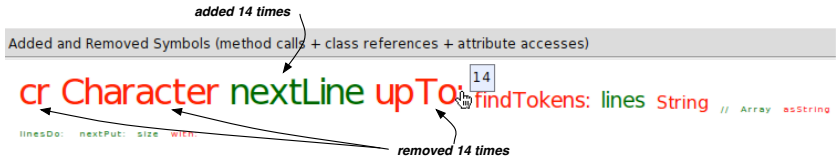

Figure 12: Replacing method calls: upTo: Character cr for nextLine.

Figure 12 shows the mixed symbolic cloud (*i.e.,* added and removed symbols) applied to a scenario where 14 methods were modified. In each method the two combined method calls upTo: and Character cr were replaced by nextLine. A user can observe on the symbolic cloud that the vocabulary involved is small even if the change is large. We present the cloud even though our experience showed that its usefulness is mitigated as we discussed in the evaluation Section.

6. Torch Infrastructure

In this section we present the infrastructure of Torch with a particular focus on the Torch underlying change and code model. Torch's core is built around a change meta-model. The Torch change meta-model is built on top of the Ring source code meta-model [43]. Torch's tools are then integrated with the versioning system Monticello to be usable in practice by Pharo developers.

6.1. Ring Meta-Model

Ring [43] is a meta-model and infrastructure for source code analysis. It provides a common API at structural and runtime level that allows existing

and new tools to interact and integrate directly with the host environment, *i.e.,* Pharo. Figure 13 shows the key definitions of the source code meta-model, which contains definitions for classes, traits[8], methods, variables (*e.g.,* instance variables, class variables), comments and containers (*e.g.,* packages).

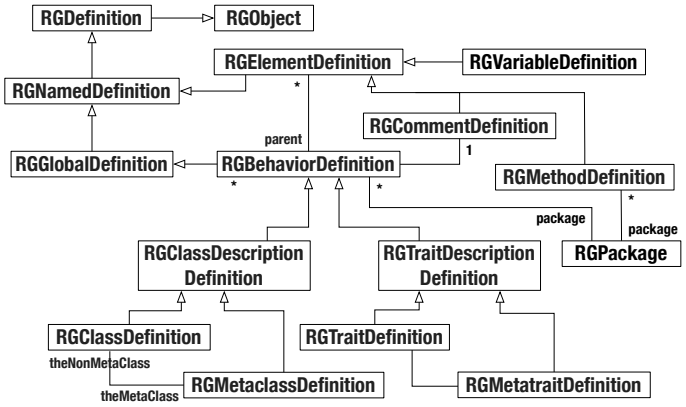

Figure 13: Ring source code meta-model – Key definitions.

The root class in the meta-model is RGObject. It allows objects to add annotations without changing their structure even at runtime. Most of the program definitions have a name, thus they inherit from RGNamedDefinition. For representing classes, traits and their meta-classes the meta-model provides the RGBehaviorDefinition class hierarchy, and for program entities defined within a class, such as variables and methods the meta-model offers the RGElementDefinition class hierarchy. Note that class comments are also first-class entities in Ring (RGCommentDefinition). Finally, the meta-model also represents container classes used for deployment and ownership representation (*i.e.,* RGPackage).

Ring unifies two kinds of relationships between several program entities: a containment relationship expressed by the *parent* link (*e.g.,* a variable and a method are contained in the class), and a packaged relationship expressed by the *package* link (*e.g.,* the class Object is packaged in the package Kernel).

6.2. Torch Meta-Model

Torch's meta-model is a change-based model built on top of the Ring source meta-model. The change-based model in Torch maintains a unique object representing a particular entity (*e.g.,* a class, a method, a package). Torch objects are stateful and know which of their properties have changed. For example, if we compare two versions in which the class Foo has changed its superclass, the versioning system provides two objects for this class containing the old and

[8] A trait is a set of methods that serves as a behavioral building block for classes. 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.

new superclass. Torch converts both objects into one TCClass object, sets the status of this object to modified, and assigns to its subclass instance variable a TCChangedElement object that contains the old and new superclass.

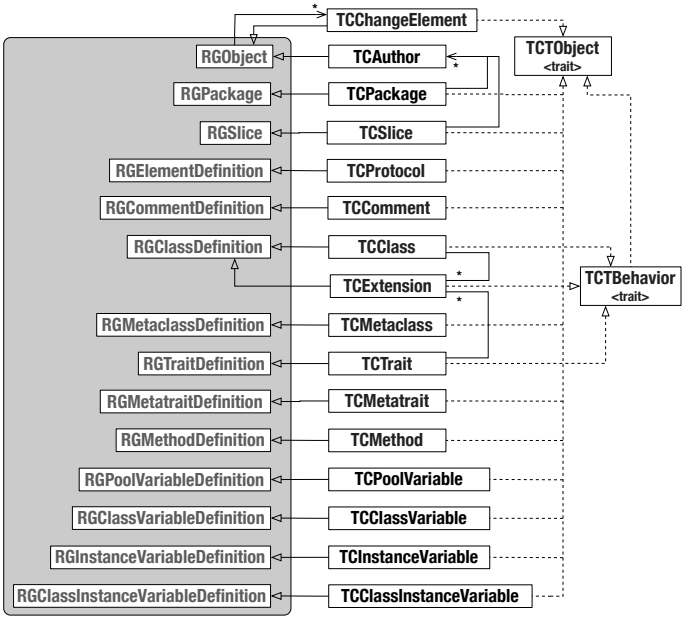

Figure 14: Torch change meta-model – Key classes.

In Figure 14 we show the Torch 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 are only showing the inheritance relationships between Torch and Ring classes. Other associations between Torch classes are semantically the same as in the Ring classes shown in Figure 13, *e.g.,* a TCPackage object may have 0..* TCMethod objects.

Currently Torch is tailored towards Smalltalk programs. Two classes have been introduced in Torch, TCProtocol and TCExtension that inherit from RGElement-Definition and RGClassDefinition respectively. TCProtocol maps the method categories, and TCExtension –a class alike definition– groups method extensions of a class per external package. In addition, several traits have been introduced in the Torch model. The TCTObject trait is used by the classes that need state and changeable properties (*i.e.,* all classes except TCAuthor). The TCTBehavior trait defines the specific behavior of classes, traits and extensions.

7. Evaluation

We gave some anecdotal evidence showing that Torch supports specific integration tasks. Our personal experience analyzing changes shows 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 for integrators 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 to use Torch on those 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[9], Pharo, Seaside[10]) from the Smalltalk community. For the second evaluation we performed a pre-experimental user study about the usability of Torch and its features with ten developers.

7.1. Field Evaluation

Table 1 shows the characteristics of the three Smalltalk projects as reported from http://www.squeaksource.com (Projects). We asked two integrators of each of these projects to use Torch during their daily work. We also asked them to answer a questionnaire[11] composed of two main parts. The first part presents closed questions that the integrators needed to mark using a 5-point Likert scale[12]. The second part consists of open questions oriented to obtain more feedback of what can be improved in Torch.

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 1: Open-Source projects in which Torch was evaluated

The six integrators (strongly) agree that change integration is a difficult task. 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 acknowledged 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 2 we present a summary of the results of four questions of the first part of the questionnaire about the general overview of the use of Torch. The full version of the questionnaire is available online at the mentioned url. The values presented in the table correspond to the number of integrators that marked a rating scale.

[9]Moose: http://www.moosetechnology.org
[10]Seaside: http://www.seaside.st
[11]Available at http://soft.vub.ac.be/~vuquilla/torch-questionnaires.zip
[12]Rating scale: strongly disagree=1, disagree=2, neither agree nor disagree=3, agree=4, strongly agree=5.

Question	Agree	Strongly Agree
Would you like to use Torch in your daily integration process?	3	3
Does the Torch dashboard help you?	3	3
Do you find the diff as a fly-by help showing code on any entity useful?		6
Do you think you got a better understanding of the changes, their scope and their impact using Torch?	3	3

Table 2: Summary of some results about the use of Torch

The table shows 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 visualization and textual diffs can be efficiently integrated.

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?
- Which features of Torch are not useful at all and should be removed? why?
- 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. We present some of them that we have considered as avenues of future work: "Torch should...": (1) merge some visualizations and provide the different representation of classes (structural and condensed) on demand, (2) display changes based on semantic impact or not, and (3) also steer the decision to merge or cherry pick a change directly from Torch and not with yet another tool.

7.2. About the symbol cloud

We did not run experience to validate the usefulness of the symbol clouds because our personal experience with it was mitigated. In rare occasions, for example when a change captured a single expression rewrite, the number and occurrence of the symbols made clear the semantics of the action. For example replacing all the occurrences of one message by another one was made clear. However, for normal development actions involving more than a couple of different expressions, the cloud just gave a vague and general impression without scope. Sorting the symbols according to their occurrence is not a really interesting attribute especially since it does not keep the semantics induced by the sequence of symbols in a piece of source code.

7.3. Pre-Experimental User Study

We performed a pre-experimental user study to assess the usability of Torch and its features with 10 participants (mostly Smalltalk developers). This kind of study is a quasi-experiment – opposed to a full scientific experiment – that does not allow us to make any absolute claim regarding the usability of Torch. However, 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 [29].

Study design. The quasi-experiment consists of a pre- and post- test. The pre-test quantifies the expectations of a developer regarding change visualization tools before using Torch. The post-test quantifies their perception of Torch and its features after applying it to two change scenarios.

We compare the results of both tests and quantify how the use of Torch impacted the developers' perception of a visual tool for understanding changes, and which of Torch's features were considered useful by the developers. Both tests used a 5-point Likert scale to score each statement – from 1 (totally disagree) to 5 (totally agree). The following properties were measured:

- Value of visualizations: Do change visualizations aid in understanding changes?
- Information usage: Does the combination of textual and graphical information speed change exploration and understanding?
- Class representation: Do UML alike 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 four steps: (1) Fill out the pre-test – 20 statements measuring their general background, attitude towards tool support and change understanding, and their expectations of a change visualization tool. (2) Attend a short presentation about Torch. (3) Use Torch to understand the second and four change scenarios presented in Section 5. (4) Fill out the post-test – 22 statements measuring their perception of the task performed and of Torch itself, their evaluation of the usage of Torch and of its different features.

Due to space limitations, we partially analyse the pre- and post-tests in this article. Both tests and the 10 filled out pre- and post-tests are available online[13].

Developer 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, experience with Smalltalk integrated development environments (*e.g.,* Pharo), usage of facilities of integrated development environments (*e.g.,* implementors of a method, method calls), and usage of facilities of revision control systems (*e.g.,* lists of changes, merge).

[13] Available at http://soft.vub.ac.be/~vuquilla/torch-developers-case-study.zip

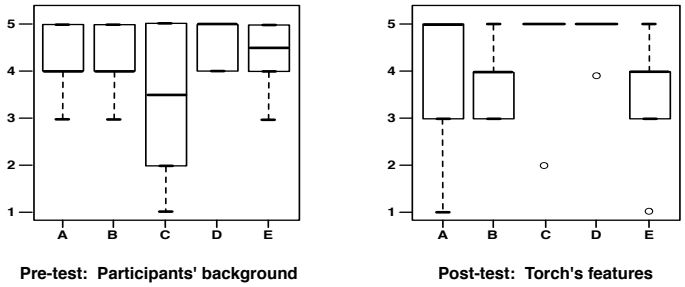

Figure 15: Boxplot of pre-test (left): (A) Development experience, (B) OO experience, (C) Smalltalk IDE experience, (D) Usage of IDE's facilities, (E) Usage of RCS's facilities. Boxplot of post-test (right): (A) Detailed class representation, (B) Package-centric visualizations, (C) Diff as a fly-by help, (D) Class structure as a fly-by help, (E) Presence of unchanged entities.

Figure 15 (left) shows a summary of this part of the pre-test. The majority of the participants qualified themselves as experienced OO developers (A) and (B). The results for (C) shows that our group of participants was not limited to Smalltalk developers. As it can be seen in (D), all of the developers highly use facilities of IDEs. Finally, they also regard themselves as knowledgeable users of revision control systems, as shown in (E).

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 16. As we can see in Figure 16(a), all the developers were very positive about the value provided by Torch, one agree and nine of them totally agree that the Torch dashboard 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.

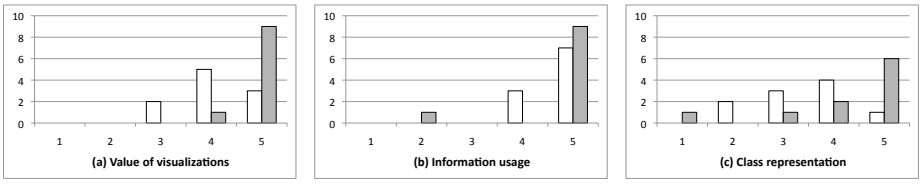

Figure 16: Comparison of the pre-test (shown in white) and post-test (shown in grey). X axis represents the 5-point Likert scale, Y axis represents the number of participants that selected a scale point.

For the *information usage* property shown in Figure 16(b) all participants (totally) agree 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 totally agree that the fly-by help offers

an advantage, confirming that combing both kinds of information enhances the support provided by Torch.

Finally, with respect to the visual representation of classes shown in Figure 16(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 (totally) agree. This perception improved after using Torch as shown in the results of the post-test. Only one participant totally disagree and one was neutral about the usefulness of a visual representation for classes similar to the one used in UML diagrams. Eight developers (totally) agree that this eased change identification.

Features of Torch. From the post-test we extracted relevant information about the usefulness of several features of Torch. The results are shown as boxplots in Figure 15 (right). The features of Torch were found very useful by the majority of participants. As you can see in the five boxplots, the median for three features corresponds to *totally agree=5* and for the other two corresponds to *agree=4*.

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 15 (right).

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 agree 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 agree with the advantages of the diff and one disagree 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.

7.4. 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.

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 change scenarios and fill out the post-test. As with the field study, we did not want to influence them when using the tool.

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 among other usage scenarios.

Perception and reality. Without having the measures of the exact time taken by the integrators while using Torch, the integrators may perceive that Torch speeds up the change exploration and understanding of changes, when the reality may be different. This is another threat to validity of this study. However, this is a subject of further study.

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 integrators because it currently supports Smalltalk programs. Since the file structure of these languages is also based on packages, classes, methods, and since the Ring and Torch meta-models can be extended we believe that the approach presented in this article 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.

7.5. 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.

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 changes, 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 would help in this situation.

- In the same vein, due to the fact that Torch allows the simultaneous comparison of multiple pair of versions, 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 packages, the edges representing inter-package inheritance relationships produce noise.

- The most important limit 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.

- Torch does not infer missing or inconsistent commit behavior (for example that a programmer changed all the methods except one). Torch does not perform any recovery of refactorings such as renaming, pull or push down methods. It just presents in a compact format the exact activities made by the developer. Such features are a future work.

8. Related Work

We mentioned in the introduction the existence of a plethora of merging techniques. However, as the focus of our work lies on supporting the understanding of changes prior to merging, and not on the actual merging itself, we do not discuss these approaches in this section.

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 [38, 37], understanding a system's structure [24], the study of evolution patterns [23, 8, 7], and the comprehension of individual classes [15, 34] and packages [14].

Beyond source code, visualizations have been proposed for other types of data such as bug tracking information [7] and versioning information [28] as well as aspect-oriented programming [20].

However, none of these approaches target aiding developers in comprehending or characterizing changes as is needed for assisting the integration of changes.

Class and method understanding. Ducasse and Lanza [15] 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.* [34]. 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 classes and methods, they do not provide the same information as Torch, that is necessary for characterizing changes between two versions. Torch eases in understanding changes regarding to the structure of the system. It could be enhanced by integrating Class blueprints and Microprints into the changed classes and methods.

Change characterization. Dragan et al. [10] 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 a three different levels of abstraction: method [11], class [13], and system [12]. Such categorization of methods (stereotypes) [11] 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.

Understanding changes. Fritz and Murphy [21] 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 pure textual and does not provide visualizations of the changes all together related to the structure of the system.

The change impact analysis tool, Chianti [32] decomposes the difference between two versions of a software project 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 is one of a large category of approaches related to test prioritization [19, 18].

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

Documenting changes. Commit 2.0 [16] 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.

Aspect analysis. Pfeiffer and Gurd [31] propose Asbro, a tool which 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 [6], 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 at. [20] 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 aspects visualization and do not support diff of source code or removal/changed code support or author information.

9. Conclusions

In this article we have presented the Torch dashboard for supporting change understanding. Torch offers change characterization, change overview, change metrics and an omnipresent contextual diff to explore not only the source code of a any method but also the complete structure of any class in the dashboard. We have described the main components of Torch and the different visualizations available. Moreover, we have also introduced the infrastructure of Torch. By means of internal experiments with typical scenarios, we have presented the capabilities of Torch for defining change context and overview. By means of external validations with actual open-source integrators, we have demonstrated that Torch is a promising tool that can definitely support integrators in understanding changes, and more important in speeding up the time they invest for integration processes. Finally, we have also performed a pre-experimental user study with ten software developers to assess the usability of Torch. We did not make absolute claims out of the results, but they have provided us with positive information about how developers perceive Torch and its features.

Acknowledgments. This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the "Contrat de Projets Etat Region (CPER) 2007-2013".

References

[1]S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. Semistructured merge: rethinking merge in revision control systems. In *ESEC/FSE*, pages 190–200. ACM, 2011.

[2]U. Asklund. Identifying conflicts during structural merge. In *Nordic Workshop Programming Environment Research*, 1994.

[3]B. Berliner. CVS II: parallelizing software development. In *USENIX*, pages 22–26, 1990.

[4]D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.

[5]J. Buffenbarger. Syntactic software merging. In *Software Configuration Management*, pages 153–172. Springer-Verlag, 1995.

[6]W. Coelho and G. C. Murphy. Presenting crosscutting structure with active models. In *AOSD*, pages 158–168. ACM, 2006.

[7]M. D'Ambros and M. Lanza. BugCrawler: Visualizing evolving software systems. In *CSMR*, pages 333–334, 2007.

[8]M. D'Ambros, M. Lanza, and M. Lungu. The evolution radar: Integrating fine-grained and coarse-grained logical coupling information. In *MSR*, pages 26 – 32, 2006.

[9]D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Trans. on Softw. Eng.*, 34(3):321–335, 2008.

[10]N. Dragan, M. Collard, M. Hammad, and J. Maletic. Categorizing commits based on method stereotypes. In *ICSM*, pages 520–523. IEEE, 2011.

[11]N. Dragan, M. L. Collard, and J. I. Maletic. Reverse engineering method stereotypes. In *ICSM*, pages 24–34. IEEE, 2006.

[12]N. Dragan, M. L. Collard, and J. I. Maletic. Using method stereotype distribution as a signature descriptor for software systems. In *ICSM*, pages 567–570. IEEE, 2009.

[13]N. Dragan, M. L. Collard, and J. I. Maletic. Automatic identification of class stereotypes. In *ICSM*, pages 1–10. IEEE, 2010.

[14]S. Ducasse, T. Gîrba, and A. Kuhn. Distribution map. In *ICSM*, pages 203–212. IEEE Computer Society, 2006.

[15]S. Ducasse and M. Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.

[16]M. DAmbros, M. Lanza, and R. Robbes. Commit 2.0. In *Web2SE: Workshop on Web 2.0 for Software Engineering*, pages 14–19. ACM, 2010.
[17]P. Ebraert. First-class change objects for feature-oriented programming. In *WCRE*, pages 319–322, 2008.
[18]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*, 2003.
[19]S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Int. Symposium on Soft. Testing and Analysis*, pages 102–112. ACM Press, 2000.
[20]J. Fabry, A. Kellens, S. Denier, and S. Ducasse. Aspectmaps: A scalable visualization of join point shadows. In *ICPC*, pages 121–130. IEEE, 2011.
[21]T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *ICSE*, pages 175–184. ACM, 2010.
[22]J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report 41, AT&T Bell Laboratories Inc, 1976.
[23]M. Lanza. The Evolution Matrix: Recovering software evolution using software visualization techniques. In *IWPSE*, pages 37–42, 2001.
[24]M. Lanza and S. Ducasse. Polymetric Views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.
[25]D. B. Leblang and R. P. Chase. Computer-aided software engineering in a distributed workstation environment. *SIGSOFT Softw. Eng. Notes*, 9(3):104–112, 1984.
[26]T. 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.
[27]E. Lippe and N. van Oosterom. Operation-based merging. *SIGSOFT Softw. Eng. Notes*, 17(5):78–87, 1992.
[28]M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The Small Project Observatory: Visualizing software ecosystems. *Science of Computer Programming, Elsevier*, 75(4):264–275, Apr. 2010.
[29]N. Matthijssen, A. Zaidman, M.-A. Storey, I. Bull, and A. van Deursen. Connecting traces: Understanding client-server interactions in ajax applications. In *ICPC*, pages 216–225. IEEE, 2010.
[30]T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.
[31]J.-H. Pfeiffer and J. R. Gurd. Visualisation-based tool support for the development of aspect-oriented programs. In *AOSD*, pages 146–157. ACM, 2006.
[32]X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA*, pages 432–448, 2004.
[33]R. Robbes. *Of change and software*. PhD thesis, University of Lugano, Switzerland, 2008.
[34]R. Robbes, S. Ducasse, and M. Lanza. Microprints: A pixel-based semantically rich visualization of methods. In *Int. Smalltalk Conference*, pages 131–157, 2005.
[35]H. Shen and C. Sun. A complete textual merging algorithm for software configuration management systems. In *COMPSAC*, pages 293–298, 2004.
[36]P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA*, pages 268–285, 1996.
[37]M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Müller. On integrating visualization techniques for effective software exploration. In *Symposium on Information Visualization*, pages 38–48. IEEE Computer Society, 1997.
[38]M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? In *WCRE*, pages 12–21. IEEE Computer Society, 1997.
[39]D. Taenzer, M. Ganti, and S. Podar. Problems in object-oriented software reuse. In *ECOOP*, pages 25–38, 1989.
[40]D. Thomas and K. Johnson. Orwell — A configuration management system for team programming. In *OOPSLA*, pages 135–141, 1988.
[41]W. F. Tichy. Rcs—a system for version control. *Softw. Pract. Exper.*, 15(7):637–654, 1985.
[42]V. Uquillas Gómez, S. Ducasse, and T. D'Hondt. Visually supporting source code changes

integration: the torch dashboard. In *WCRE*, pages 55–64, 2010.

[43]V. Uquillas Gómez, S. Ducasse, and T. D'Hondt. Ring: a unifying meta-model and infrastructure for smalltalk source code analysis tools. *Computer Languages, Systems and Structures*, 38(1):44–60, 2012.

[44]B. Westfechtel. Structure-oriented merging of revisions of software documents. In *Int. work. on softw. configuration management*, pages 68–79. ACM, 1991.

[45]N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Softw. Eng.*, (12):1038–1044, 1992.