HAL
open science

Visually Supporting Source Code Changes Integration: the Torch Dashboard

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

To cite this version:

Verónica Uquillas-Gomez, Stéphane Ducasse, Theo d'Hondt. Visually Supporting Source Code Changes Integration: the Torch Dashboard. Working Conference on Reverse Engineering, Oct 2010, Boston, United States. inria-00531508

HAL Id: inria-00531508

https://inria.hal.science/inria-00531508v1

Submitted on 2 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Visually Supporting Source Code Changes Integration: the Torch Dashboard

Accepted to WCRE 2010

Verónica Uquillas Gómez*†, Stéphane Ducasse†, Theo D'Hondt*
* Software Languages Lab — Vrije Universiteit Brussel — Belgium
Email: {vuquilla | tjdhondt}@vub.ac.be
† RMoD Team — INRIA Lille-Nord Europe Research Center
Laboratoire d'Informatique Fondamentale de Lille — Université Lille1 — France
Email: stephane.ducasse@inria.fr

***Abstract*—Automatic and advanced merging algorithms help programmers to merge their modifications in main development repositories. However, there is little support to help *release masters* (integrators) to take decisions about the integration of published merged changes into the system release. Most of the time, the release master has to read all the changed code, check the diffs to build an idea of a change, and read unchanged code to understand the context of some changes. Such a task can be overwhelming. In this paper we present a dashboard to support integrators getting an overview of proposed changes in the context of object-oriented programming. 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. We describe our experiment applying it to Pharo, a large open-source system, and report on the evaluation of our approach by release masters of several open-source projects.**

This paper makes heavy use of colors. Please read a colored version of this paper to better understand the presented ideas.

I. Supporting Change Integration

Integrating changes that represent fixes, enhancements or new features are key software development activities. Still there is no adequate support to help integrators. Current state of the art are mostly textual diff tools, and there is one diff tool based on ASTs. In particular, integrators do not get an overview of the changes (how changes are distributed, what groups of entities changed) and at the same time the possibility to understand detailed changes within their specific context.

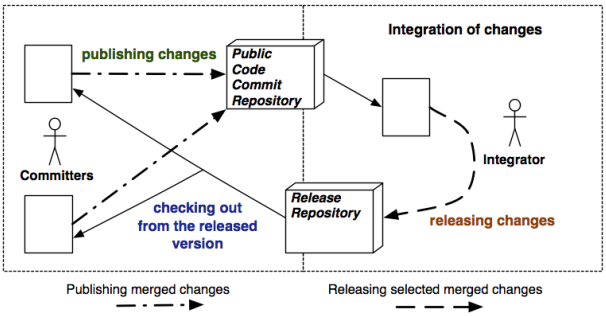

Fig. 1. Different roles and actions in change integration chain

Figure 1 shows the conceptual process of changes integration as it can be found in open-source projects. 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 (acceptation testing and others), we can identify two roles: committers of changes and integrators of such changes.

- *committers* checkout code from a public released version – but it could be from another alpha or beta version of the current development. They commit their changes (fixes, enhancements) in a conceptual repository[1].
- *integrators* integrate the code of *committers* by merging the changes made to previous versions into the current (and future released) version.

There are several merging techniques: text-based [1]–[3], syntactic-based [4]–[6], semantic-based [6], [7], operation-based [8], [9] and merging algorithms such as 2-way merge [10] and 3-way merge [11]. Several tools such as Envy [12] take into account the underlying meta-model as a step towards a semantic merge. Still, integrating changes is a difficult task. Indeed, 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 paper, we present *Torch*, a dashboard displaying object-oriented[2] changes using different visualizations and change summaries. In particular, Torch provides several change overviews based on packages or class hierarchies visualizations while at the same time it offers *an omnipresent contextual diff based on a fly-by-help*. In addition, integrators can use two different scenarios: they can focus on the changes by getting a delta based either on the version in which those changes were made (*i.e.,* ancestor), or on the context of the latest version of head or trunk (of the release stream) using a 3-way merge which takes into account the common ancestor version. See both scenarios in Figure 2.

The paper is structured as follows. In Section II we present the challenges of characterizing changes. Torch is introduced

[1]The distinction between commit and release repositories is conceptual since tagging mechanisms and tool support can distinguish between them.

[2]The current version of Torch analyzes systems implemented in Smalltalk.

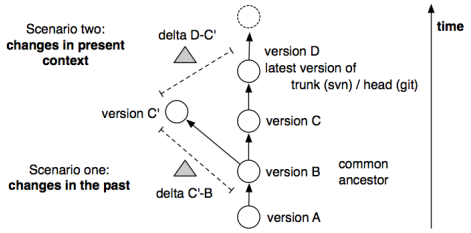

Fig. 2. Changes in the past and within trunk context.

in Section III, followed by a detailed definition of the different visualizations in Section IV. We illustrate the use of our approach in Section V using seven examples. Afterwards, we present a discussion of the evaluation of Torch in Section VI. Section VII is dedicated to discuss related work, and we conclude this paper in Section VIII.

The contributions of this paper are: (1) identification of integrator needs and (2) a change dashboard, named *Torch*, that aims at helping developers and integrators to understand changes.

II. Changes Characterization Challenges

Patches are changes of source code that do not track the complete history of actions that led to the changes [13], [14]. From that perspective, operation-based merge [9], [15] is ruled out since it is based on the idea that either refactorings or every single action made by the programmer is fully recorded.

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 3-way merge (*i.e.,* common ancestor is taken into account to support automatic merging and conflict resolution) [11]. Diff tools that show the changes as code snippets can be used easily 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 paper was one of the integrators of the Squeak open-source project. Since two years now, he is one of the integrators of the Pharo open-source project. From this experience we assemble a non-exhaustive list of challenges which integrators face daily. Note that our solution does not solve all the problems presented here but we believe that it constitutes a reference for change integration challenge identification.

- *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 through a complete system.
- *Kind of Actions*. Understanding whether the changes are mainly adding, removing or changing 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 and not (*e.g.,* just license or comments) is complementary to the other information.
- *Vocabulary*. In certain situations, assessing the difference in vocabulary between a change and its 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.
- *Architectural Drifting*. When integrating a change, it is also 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 as shown in Figure 2.
- *Impact of the changes*. A much more difficult problem is to understand the impact of a change. This is particularly difficult in presence of the *fragile base class problem* [16]. 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.

Our approach in a nutshell: To support change integration, our approach characterizes changes according to change structure, author and symbolic information. The integrator can see contextual diffs representing the changes using class hierarchy or package centric visualizations that show classified changes.

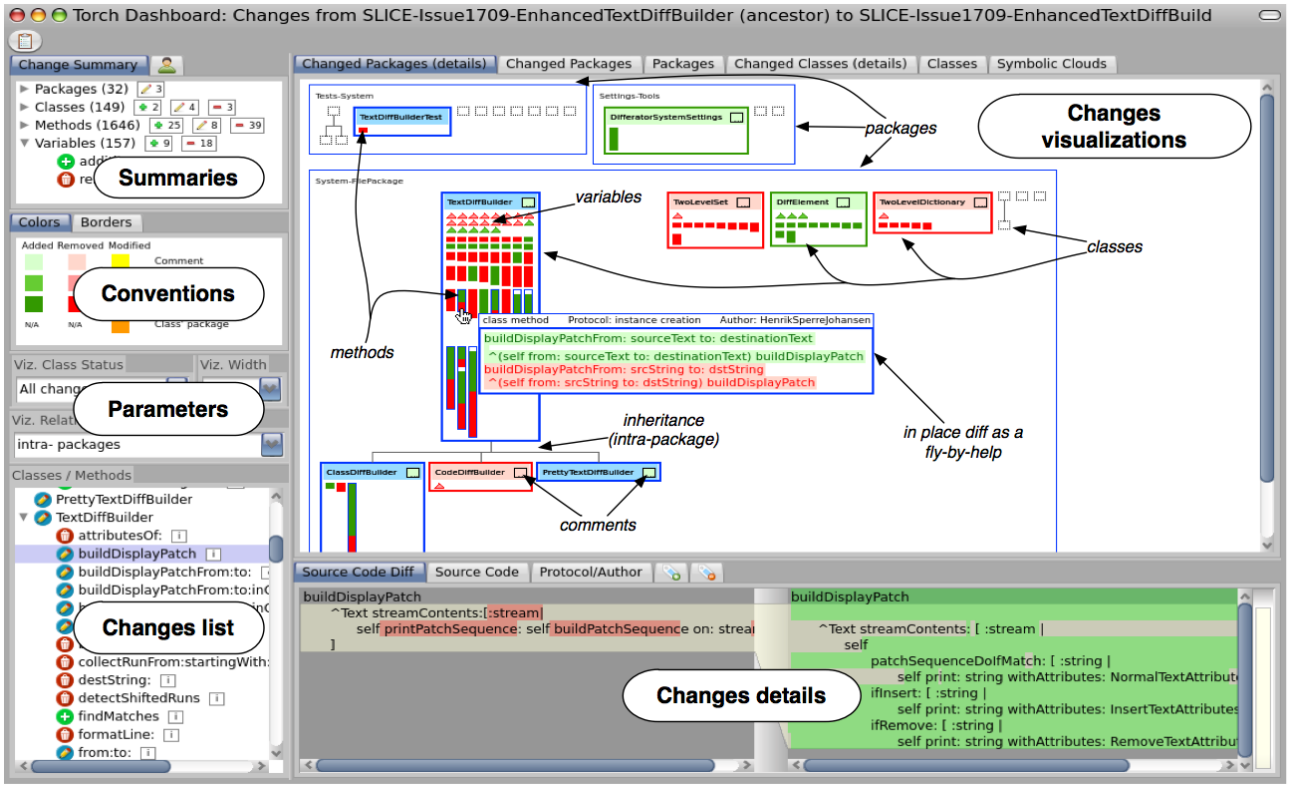

Fig. 3. Dashboard main elements: the *summaries* give an idea of the size of the changed entities and the actual changes; the *changes list* presents the list of changes and their detailed difference using the *changes details*; the *changes visualizations* present a map of changes structured around packages and classes.

III. Supporting Change Understanding in Early Integration Phase with Torch

Our approach, *Torch*[3], provides visual tool support to integrators to characterize and understand changes in context. Torch presents metrics about changes per entity and per author, and a contextual diff view on top of the visualizations. Based on the interview with the release masters of open-source projects, and a literature survey we identified the information that can help characterize changes and address some of the challenges previously identified.

The dashboard presents different structural representations of changes using visualizations. It speeds up the access to their textual information using a diff as a fly-by-help. By combining graphical and textual information, Torch brings semantic information to changes exploration. The visual mapping of changes to their structural representation helps integrators to get a quick overview of the changes and to understand some of their characteristics, such as scope, size, type of change, vocabulary involved or 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.

Figure 3 shows the structure and main elements of the dashboard. Due to space limitations, in this paper we only present an overview of each element. In Section IV we present a more detailed description of the *Changes visualizations* element of the dashboard.

- *Summaries*. Summaries 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). This information is shown per entity (*i.e.,* packages, classes, methods, variables) and per type of change (*i.e.,* added, modified, removed) in the first summary. A second summary presents measures of changes per user and per type of change since it may be important to understand who was responsible for the changes.
- *Conventions*. Colors are used to represent entities and types of changes. They are always visible to help the user to get instantaneous information and reinforcement of his knowledge. The conventions are the same in the entire dashboard: green for additions, blue for modifications, red for removals, and yellow for comment modifications. 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 applied to any entity. Integrators can parameterize which classes should display their data by means of the *class status* parameter (*i.e.,* added, modified, removed, unchanged). Inter-package relationships can be

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

displayed by demand using the *relationships* parameter.

- *Changes list*. Classes and methods representing changes are shown in a list, which can be generated based on any measure or visual entity.
- *Changes details*. Method source code, class definitions, comments, authors, protocols, and symbols (*i.e.,* vocabulary involved) are presented mainly using a diff. Such information is available through the *Changes list*.
- *Changes visualizations*. This is the main element of the dashboard, where unchanged and changed entities with their structural representations are mapped to visual entities. The changes are highlighted respecting the *Conventions*. A contextual fly-by-help supports an in place diff view. Figure 3 shows three packages containing classes.

Torch is developed in Smalltalk and it is available for Pharo[4]. Torch relies on the information provided by the Monticello SCM system. To give integrators direct access to versions (or group of versions – slices) in a Monticello repository, Torch is integrated with the Monticello tools.

IV. Dashboard visualizations in details

Version comparison is graphically presented in the central element of the dashboard, named *Changes visualizations*. There, packages, classes, methods, variables and their relationships, and the vocabulary involved in changes are visually presented. Optionally 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.

Changes information can be displayed according to a package-centric level (as shown in Figure 3) or from a system point of view based on an inheritance tree (*i.e.,* class-centric level as shown in Figure 13). Note that inheritance relationships are also drawn within packages since those relationships add semantics and provide information about the impact of a change as shown in Figure 9 and in 13.

Table I summarizes the two kinds of visualizations offered by Torch, namely package-centric and class-centric. For each kind of visualization, the table presents which entities are displayed and which visual representation is shown for classes.

Package-centric		**Class-centric**	
Packages	*Class representation*	*Classes*	*Class representation*
changed	structural	changed and dependents	condensed
changed and unchanged	condensed	changed and unchanged	structural

TABLE I
Torch Dashboard Visualizations

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" [17]. 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, we explain the visual representation of entities and the fly-by-help utility.

A. Entities Representation

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

Packages: A package is displayed as a large box containing all its classes (even unchanged ones). Inside, when possible, classes are organized in class hierarchies, and they show changes at structural or condensed representation (explained later). Unchanged classes are represented by small dashed boxes. Figure 4 shows the modified System-FilePackage and its changed classes using structural representation.

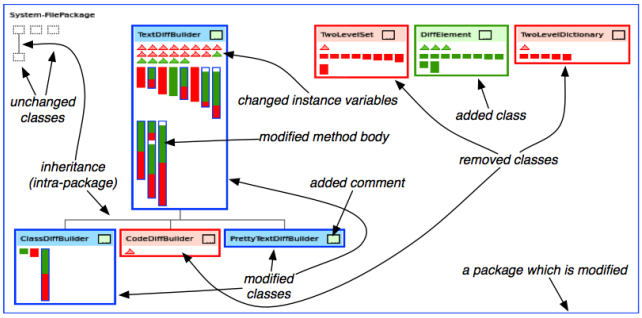

Fig. 4. Package containing unchanged classes (small dashed grey boxes), removed (red boxes), added (green box) and modified classes (blue box). Classes contain variables (triangles) and methods (bars).

Classes: A class has two visual representations for its changes: structural representation and condensed representation. Table I indicates which visualization is using the structural or condensed class representation[5]. In Figure 5 we show three classes (added, removed and modified class) and for each we show both representations. The colors of the border and the background of the class name represent whether the class was entirely added (green/light green), removed (red/light red) or simply modified (blue/light blue). Our class representations may also display a colored box beside the class name for *changed class comment*.

- *Structural*. A class is displayed using sections: class name section, attributes section and methods section (first three classes in Figure 5). Note that attributes and methods appear depending on the *class status* parameter (by

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

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

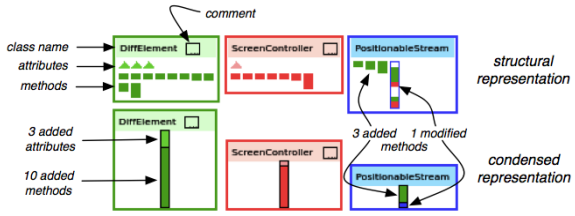

Fig. 5. Structural and condensed visual representation of classes

default *changed*, therefore only changed attributes and methods are presented). DiffElement and ScreenController have changed attributes, methods and comment, whereas PositionableStream only has changed methods and does not show the attributes section. 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 them (see last three classes in Figure 5). The bar is composed of colored segments. Each segment groups changes (*e.g.,* removed method, added attributes), uses a color for that group of changes (*e.g.,* added methods in dark green, modified methods in blue) and sets a height (the number of those changes). This visual representation also includes a class name section as the *Structural* representation.

B. 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 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 6 shows a diff as a fly-by-help. It shows a method's code and highlights line additions in red and removals in green. The background color of changed lines is also set to light green and light red. This allows us to show empty lines additions or removals. In addition, a line above the source code presents extra information of a method: the scope (*i.e.,* instance or class method), the protocol and the author.

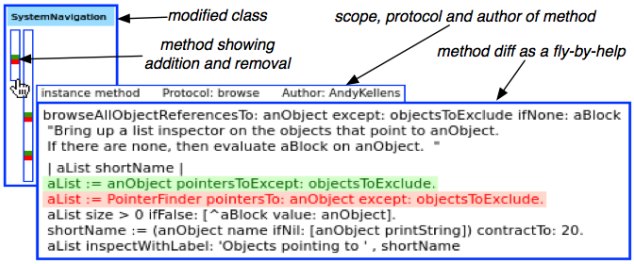

Fig. 6. Omnipresent code browsing: diff as a fly-by-help

Full class structure as a fly-by-help: Most of our visualizations that present classes only include *changed* methods and variables. Torch complements this information by also offering a fly-by-help over the class name that shows the *full* class structure, shown on the right in Figure 7. Integrators see unchanged methods and variables 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 integrators to observe the structure of any class in the dashboard.

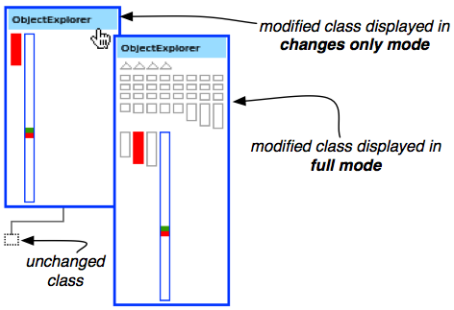

Fig. 7. Class displayed in changes only mode on the left, and in the full mode as a fly-by-help on the right

C. Package-centric Visualizations

Package-centric visualizations provide the structural context of any existing change, by distributing classes among 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 process of changes.

- *Changed Packages (details)*. When comparing versions with many unchanged packages, this visualization decreases its size and complexity by presenting only 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 variables, 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 presenting also unchanged packages. Classes use 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.

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 13.

D. Symbolic Clouds

Torch presents a third kind of visualization, named Symbolic Clouds. They show the vocabulary involved in changes instead of changed entities (at version, package, class and method level). The purpose of Symbolic Clouds is to give hints to integrators 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).

Currently, Torch collects method invocations, class references, and access to instance variables and to three literals values (*i.e.,* nil, true, false) from changed source code to construct the symbolic clouds. Each symbol is associated with the number of its occurrences in the source code and with a color defined in the conventions. The number is mapped onto a font size that is used for drawing that symbol.

Three symbolic clouds are aggregated to convey the added, removed and mixed symbolic information.

- *Added symbolic cloud*. It shows added symbols in green. They are sorted based on their occurrences which draws the attention to the first symbols that represent the most frequently added symbols in the source code.
- *Removed symbolic cloud*. It shows removed symbols in red. The purpose of this cloud and the order of symbols in it are exactly the same as in the cloud for added symbols.
- *Mixed symbolic cloud*. It combines added and removed symbols *sorting them alphabetically*. This emphasizes the same symbols that were added and removed in different methods (as they appear next to each other in green and red respectively).

Figure 8 shows the two first clouds applied to the scenario where the method calls = nil (red symbols) were replaced by the method call isNil (green symbol). Figure 15 shows another example usage based on the mixed symbolic cloud.

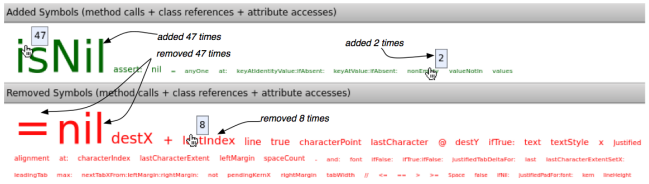

Fig. 8. Added and removed symbolic clouds

V. Torch at Work

Torch is used by several open-source projects (Moose, Pharo, Seaside). Here we illustrate Torch by applying it to the changes streams of the Pharo project (versions 1.0 and 1.1). We took two repositories http://www.squeaksource.com/PharoInbox containing the submissions and http://www.squeaksource.com/PharoTreatedInbox containing the submissions once they have been integrated into the current release. We present 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.

Here is a list of scenarios:

- Removing a feature and deprecating its API
- Removing a feature
- Introducing a feature
- Pushing up methods in a class hierarchy
- Introducing a method in a class hierarchy
- Editing comments
- Replacing a method call

A. 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 9 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.

B. Removing a feature

Developers that are cleaning dead code usually propose feature removals. An integrator can easily detect a feature removal in the Torch dashboard. The pattern is simple (*i.e.,* mostly removed entities) but it can be subtle: indeed clients may have to be changed not to refer to the removed features. The dashboard provides a broader view than a list of changes, it shows the magnitude and impact of such a removal on the system built using the structure of its entities.

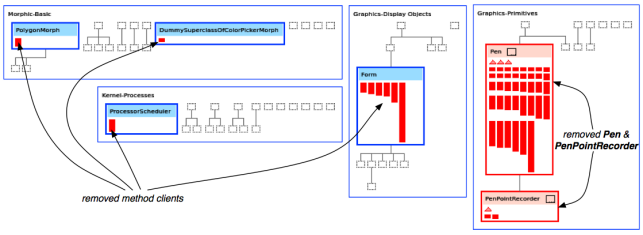

Fig. 10. Removing the feature *Pen*: classes Pen and PenPointRecorded were removed and their client classes also removed entire methods.

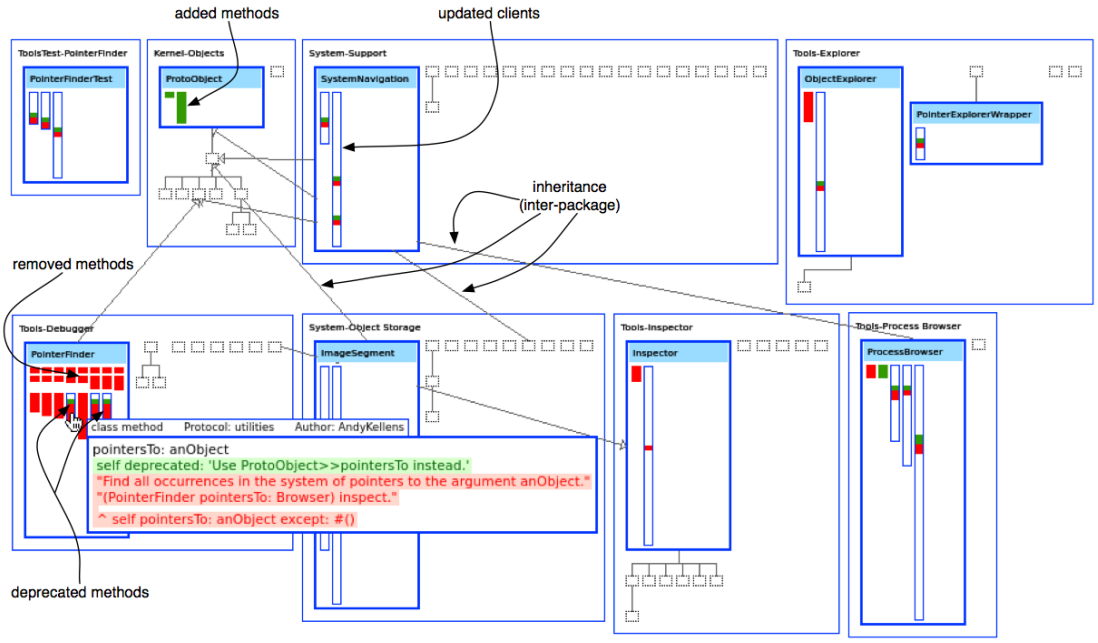

Fig. 9. Removing the feature PointerFinder and deprecating its API: its functionality was substituted by another tool.

Figure 10 shows the removal of the feature *Pen*. The classes Pen and PenPointRecorder were completely removed (all their methods are red and the class borders are red as well indicating that the classes have been removed), as well as their client methods.

Figure 11 shows a slightly different situation. The feature *SuperSwikiServer* was removed but some client methods got adapted (*i.e.,* modified) by removing a few lines of code.

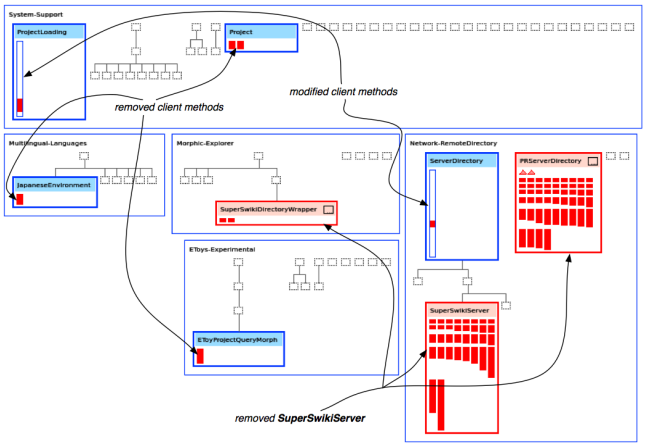

Fig. 11. Removing the feature SuperSwikiServer: two methods in clients were *modified* and other methods were simply removed.

C. Introducing a feature

The introduction of a feature is often characterized by the addition of complete classes as well as new method definitions, and some modifications introducing that feature in existing classes. When a feature introduction is submitted as a single set of changes, it is easily identified in the dashboard.

Figure 12 shows the introduction of the PopupChoiceOrRequestDialogWindow (user interface). The box of that class has green border to represent the fact that it was not modified but added. We see that a class has been added as a hierarchy leaf and that some methods have been added to other packages. Browsing the code with the fly-by-help confirms that different UI builders (PSUIMManager, UIManager and UITheme) provide access to the new feature.

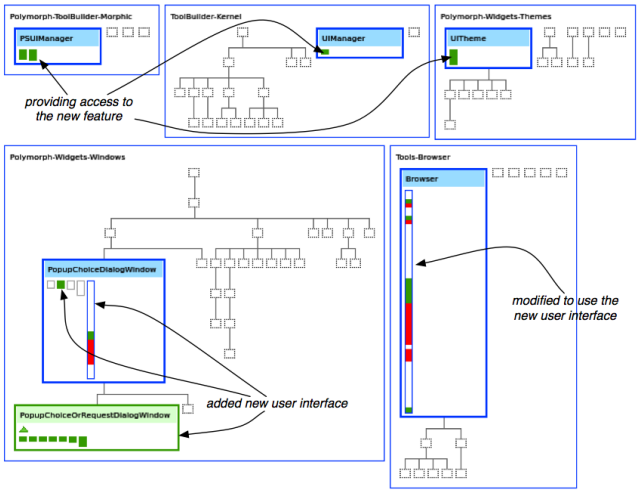

Fig. 12. Introducing a feature: a new user interface

D. Pushing up methods / Introducing a method 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 13 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.

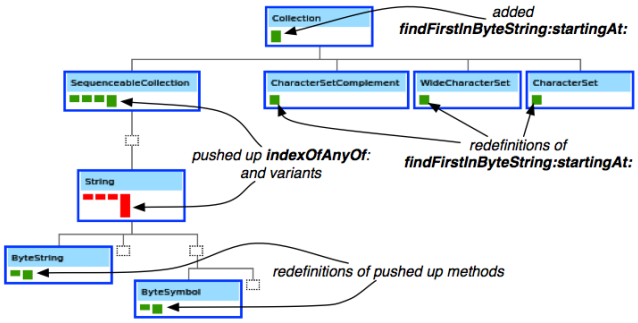

Fig. 13. Pushing up methods in the SequenceableCollection class hierarchy, and introducing a method in the Collection class hierarchy.

E. Editing comments

Cosmetic changes, such as edits in class comments are often useful and do not change the behavior of an application. In addition they can be distributed over many entities producing large lists of changes. Using a diff tool, an integrator will check each change just to find whether it was a cosmetic change, using valuable time for a task that should not demand it. When a class comment is changed the comment box beside the class name is displayed in yellow. The integrator will know that even though a change can be large in number of modified entities, there is no complexity.

Figure 14 shows the modification of comments in many classes defined in the core of Pharo: 64 classes among 41 packages were modified. The goal of these changes was to remove a reference to the word squeak, the ancestor of the system. What the figure shows is that integrators can focus on the modifications that were not class comment edition.

F. Replacing a method call

Defining new functionality that optimizes existing functionality is common in software evolution. Applying that new functionality may be represented as a set of modified methods that contain replaced method calls. 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 integrators as

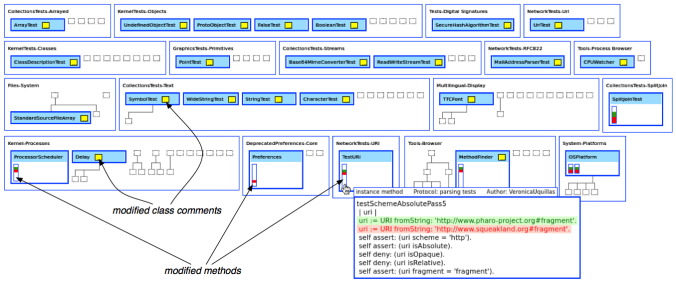

Fig. 14. Editing comments: for removing the *squeak* word

they will probably inspect every change. Torch provides three variants of symbolic clouds that aim to show the most relevant vocabulary involved in a change, and for this scenario the symbolic clouds fit perfectly as they will show few symbols (referring to method calls) with a high occurrence.

Figure 15 shows the mixed symbolic cloud (*i.e.,* added and removed symbols) applied to a scenario where 20 methods were modified. In each method the two combined method calls at:put: and at:ifAbsent: were replaced by at:ifAbsentPut:. An integrator can observe that there is few vocabulary involved and each call in this case has a high occurrence as observed on the cloud.

Fig. 15. Replacing method calls

VI. Evaluation

The previous section illustrates how Torch supports specific integration tasks. Our personal experience analyzing changes shows us that Torch helps integrators 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 rather difficult to perform a controlled experiment with master or PhD students since we need experts (release masters) of complex systems. In addition, accessing a large number of integrators is nearly impossible since integrators are busy people who are difficult to reach. We approached the integrators of three projects (Moose[6], Pharo, Seaside[7]) that we know in the Smalltalk community and we performed a limited field study.

Project	Packages	Classes	Methods	LOC	Versions	Downloads
Moose 4.x	55	568	11190	68828	2104	135434
Pharo 1.x	103	1835	46776	350266	6533	888034
Seaside 3.0	79	1017	14933	99435	4038	346063

TABLE II
Open-Source projects in which Torch was evaluated

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

A. Field Evaluation

Table II shows the characteristics of the three Smalltalk projects as reported from http://www.squeaksource.com (Projects). We asked two release masters of each of these projects to use Torch during their daily work. We also asked them to answer a questionnaire[8].

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 III we present a summary of the results of four relevant questions about the general overview of the use of Torch. The full version of the questionnaire is available online at the mentioned url. The values correspond to the number of integrators that marked such a scale[9].

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. It confirms that integration is a textual activity but that visualization and textual diff can be efficiently integrated.

The questionnaire also 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.

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..."

- merge some visualizations and provide the different representation of classes (structural and condensed) on demand.
- display changes based on semantic impact or not.
- also steer the decision to merge or cherry pick a change directly from Torch and not with yet another tool.

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

- show inheritance 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.

[8]Available at http://soft.vub.ac.be/~vuquilla/torch-questionnaires.zip

[9]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 III
SUMMARY OF SOME RESULTS ABOUT THE USE OF TORCH

B. Threats to validity

We did not interact personally with the integrators to avoid any bias. They were provided with the Torch tool, a short tutorial and the questionnaire files.

Language generalization: The integrators who replied to our validation are all Smalltalk programmers. We did not test Torch on Java or C# programs and programmers. Since the file structure of these languages is also based on packages, classes, methods, we believe that Torch should work on such 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.

This experiment only targeted integrators but we also plan to target committers so that they can understand and control their changes before publishing them. In addition, Torch will be integrated in Pharo 1.2. (version in progress) and we will perform a larger evaluation with both kind of users.

C. Discussions

While doing our internal validations of Torch we have detected some possible scenarios that decrease the level of help provided to integrators:

- When commits are messy and contain several unrelated code, Torch presents the situation as it is and it does not support tagging to classify changes. Being able to tag changes into kind of slices would help in this situation.
- In the same vein, the simultaneous comparison of a high number of versions may result in a complex visualization due to the number of drawn entities and inheritance relationships. If changed classes have a considerable number of subclasses and if those subclasses are defined in different packages, the edges representing inter-package inheritance relationships produce noise.
- The most important limit of Torch is that Torch only shows structural information and as an integrator understands the impact in terms of *different program behavior* is also very important. Of course, assessing the impact of a change on the program behavior is a really difficult task since it is another step towards semantic merge or understanding program semantics. Still we believe that it

should be possible to give an estimate about the number of impacted behavior using slicing analysis.

VII. Related Work

As mentioned in the introduction, there exists a plethora of merging techniques. 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 detail here.

In the AOSD field, several tools present aspects and how they crosscut systems. The AspectJ Development Toolkit (AJDT) for Eclipse [18] is arguably the most mature toolkit for Aspect-Oriented Programming. It offers a visualization tool which is a continuation of the AspectBrowser work by Shonlhe et al. [19]. It offers a Seesoft-inspired view [20].

Zhang et al. [21] present an analysis toolkit for assessing the impact of structural modifications through AspectJ inter-type declarations on the behavior of the system. They propose analyses to assess how the declarations impact the method lookup of the base program, and to identify how particular base-code entities are shadowed by inter-type declarations. An integration with Eclipse presents the results using markers and dedicated views that represent the lookup impact and shadowing impact.

Horie and Chiba [22] provide information on structural modifications and extended module interfaces, respectively. However they use a textual tree-based representation to show the data which limits the overview aspects. Pfeiffer and Gurd [23] propose 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 [24], 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. However, Asbro as well as ActiveAspect are dedicated to aspects visualization and do not support diff of source code or removal/changed code support or author information.

Commit 2.0 [25] 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.

VIII. Conclusions

In this paper we have presented the Torch dashboard. It offers change characterization, change overview as well as an omnipresent contextual diff. 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.

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] W. F. Tichy, "Rcs—a system for version control," *Softw. Pract. Exper.*, vol. 15, no. 7, pp. 637–654, 1985.

[2] B. Berliner, "Cvs ii: parallelizing software development," in *USENIX*, 1990, pp. 22–26.

[3] D. B. Leblang and R. P. Chase, "Computer-aided software engineering in a distributed workstation environment," *SIGSOFT Softw. Eng. Notes*, vol. 9, no. 3, pp. 104–112, 1984.

[4] J. Buffenbarger, "Syntactic software merging," in *Software Configuration Management*. Springer-Verlag, 1995, pp. 153–172.

[5] U. Asklund, "Identifying conflicts during structural merge," in *Nordic Workshop Programming Environment Research*, 1994.

[6] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Int. work. on softw. configuration management*. ACM, 1991, pp. 68–79.

[7] D. Binkley, S. Horwitz, and T. Reps, "Program integration for languages with procedure calls," *ACM Trans. Softw. Eng. Methodol.*, vol. 4, no. 1, pp. 3–35, 1995.

[8] H. Shen and C. Sun, "A complete textual merging algorithm for software configuration management systems," in *COMPSAC*, 2004, pp. 293–298.

[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.*, vol. 34, no. 3, pp. 321–335, 2008.

[10] J. W. Hunt and M. D. McIlroy, "An algorithm for differential file comparison," AT&T Bell Laboratories Inc, Tech. Rep. 41, 1976.

[11] 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.

[12] D. Thomas and K. Johnson, "Orwell — A configuration management system for team programming," in *OOPSLA*, 1988, pp. 135–141.

[13] R. Robbes, "Of change and software," Ph.D. dissertation, University of Lugano, Switzerland, 2008.

[14] P. Ebraert, "First-class change objects for feature-oriented programming," in *WCRE*, 2008, pp. 319–322.

[15] E. Lippe and N. van Oosterom, "Operation-based merging," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 5, pp. 78–87, 1992.

[16] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt, "Reuse contracts: Managing the evolution of reusable assets," in *Proc. of OOPSLA '96*, 1996.

[17] N. Wilde and R. Huitt, "Maintenance support for object-oriented programs," *IEEE Transactions on Softw. Eng.*, no. 12, pp. 1038–1044, 1992.

[18] A. Colyer, A. Clement, G. Harley, and M. Webster, *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools*, 2004.

[19] M. Shonle, J. Neddenriep, and W. Griswold, "Aspectbrowser for eclipse: a case study in plug-in retargeting," in *eclipse '04: Proc. of OOPSLA workshop on eclipse technology eXchange*. ACM, 2004, pp. 78–82.

[20] S. G. Eick, J. L. Steffen, and S. Eric E., Jr., "SeeSoft—a tool for visualizing line oriented software statistics," *IEEE Trans. on Softw. Eng.*, vol. 18, no. 11, pp. 957–968, 1992.

[21] D. Zhang, E. Duala-Ekoko, and L. Hendren, "Impact analysis and visualization toolkit for static crosscutting in aspectj," in *ICPC*, 2009.

[22] M. Horie and S. Chiba, "Aspectscope: An outline viewer for aspectj programs," *Journal of Object Technology*, vol. 6, no. 9, pp. 341–361, 2007.

[23] J.-H. Pfeiffer and J. R. Gurd, "Visualisation-based tool support for the development of aspect-oriented programs," in *AOSD*. ACM, 2006, pp. 146–157.

[24] W. Coelho and G. C. Murphy, "Presenting crosscutting structure with active models," in *AOSD*. ACM, 2006, pp. 158–168.

[25] M. D'Ambros, M. Lanza, and R. Robbes, "Commit 2.0," in *Web2SE '10: Proc. of the 1st Workshop on Web 2.0 for Softw. Eng.* ACM, 2010, pp. 14–19.