Smart Differ - A better tool for code review

Arul Siva Murugan Velayutham, Ramakrishna Rajanna, Devesh Yamparala

Abstract— Understanding and reviewing code changes will need the context of the existing code and how the change is going to affect the new call flow. Current code reviews are based textual file diff. They do not have any language semantic information of changed code. This paper focuses on using some semantic information and present an alternative code diff visualization to help the review process.

Index Terms— Code review, Code changes, Software Engineering, Code Understanding, Code Learning, Code Flow, Code diff

1 INTRODUCTION

Currently file based textual differ[1][2] is provided by code-review tools. The file based differs provide an excellent mechanism to do code reviews. However reviewers face still difficulties in

- Identifying where to start the review: In the change-list where is the logical start of the changed code and how the changes are related cannot be shown with the current tools. They just list the files in lexicographical order for review. Reviewer has to create and remember a mental model of the new control flow.
- Identifying refactoring code: Simple move detection within files can be shown in current tools but detecting code moves (refactoring) across files or unfaithful (slight modifications) within files is not possible with the current tools.
- Navigating though changes: Navigation with the change-list is tough when lots of files present in the change. It gets tougher for the reviewer when he wants to switch between files inside and outside the change-list multiple times.
- Looking at similar changes together: When a method signature has changed, reviewer would like to look at all the callers to check the usage.

In this paper we present call-graph based smart differ and show how alternative visualization can improve the code review process.

Call-graph based differ shows the diff between the call-graphs for the two versions of the change-list. The nodes are methods / functions, which have changed. The nodes show inline diff of the changed code. The edges are color coded to differentiate new / deleted calls in the change-list. The call-graph will only show the changed (new / deleted / modified) nodes (methods).

Code-bubbles kind of navigation can be used to expand the call graph to look at unchanged code in the same view.

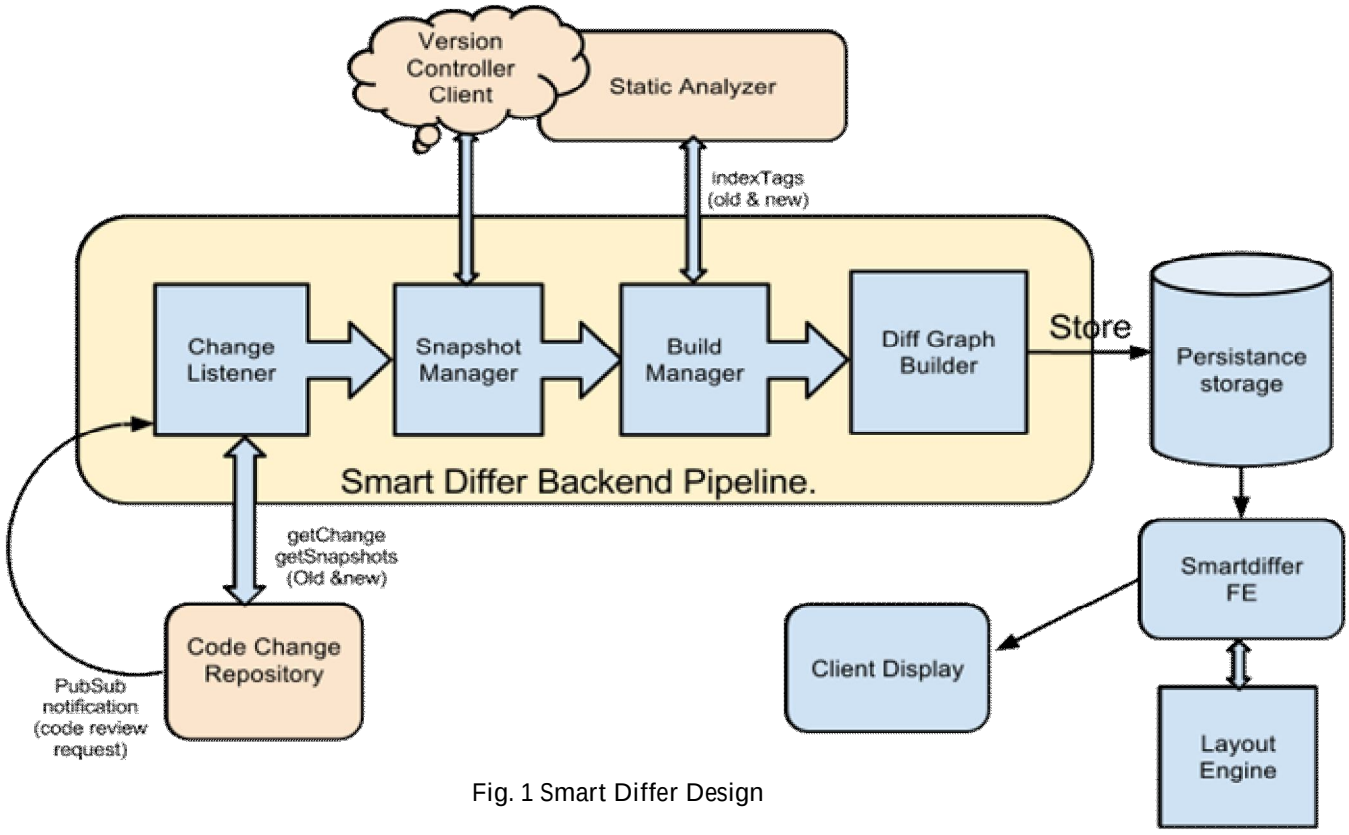

Fig. 1 Smart Differ Design

2 Design of Smart Differ

There are two parts to Smart differ service:

2.1 Backend pipeline

Listens for Code Change review requests and analyzes the changes to constructs call-graphs for base version and changed version of the code. Subsequently it merges the call graphs of base version and changed versions with node and edge annotations. It generates a layout of the merged call-graph for the presentation and stores the data to persistence storage.

2.2 Frontend

The merged call-graph is fetched from data store; layout is computed and rendered via the Smart Differ front-end.

3 Architecture of Pipeline

The Smart Differ Back-end Pipeline had been implemented as work-flow model; (i.e) the components are split in such a way that once task has been completed at one component it is pushed to the next in the pipeline (fig. 1). Pipeline has the following components arranged in that order

1. Change Listener
2. Snapshot Manager
3. Build Manager
4. Diff Graph builder

3.1 Change Listener

The Change listener(s), implements a Goops [3] subscriber for change-list creation notifications.

On receiving the notification, the change is filtered based on the following criteria:

Changes with more than threshold delta lines (at least 10 lines) of code (Graph visualization is less useful than text diff for small changes) in the programming language files. The filtered changes are passed to the next component: Snapshot Manager

3.2 Snapshot Manager

This stage creates two code snapshots reflecting the base version and the changed version of files in the change-list. The two versions are needed to construct the respective call graphs. Snapshot manager also handles the file line difference and generates the data needed for the in-line diff that required in the later part of the pipeline.

3.3 Build Manager

Build tool (similar to ctags [4]) **is slightly modified to do the** static analysis of the code. Build manager also identifies the targets that would get affected by the current files that are changed. The modified build tool is used to build Graph Index that is similar to ctags output for both code snapshots.

The generated index will have the set of nodes, each node will have properties like location, fully/partially qualified name, type etc. Type of the node will tell if its a method/function, keyword, signatures, parameters etc. The tool also generates the associations between nodes. Association represents relationships like caller-callee, inheritance etc between two nodes.

3.4 Diff Graph Builder

At this stage changed files are analyzed to get the changed methods (only nodes that are of type methods and are changed or added or deleted) & their "caller-callee" associations. With the collected set of nodes and associations the call graph is generated individually for the old and new revision of the methods that were changed. Then an in-line diff of the two revisions of the method contents are generated with lines

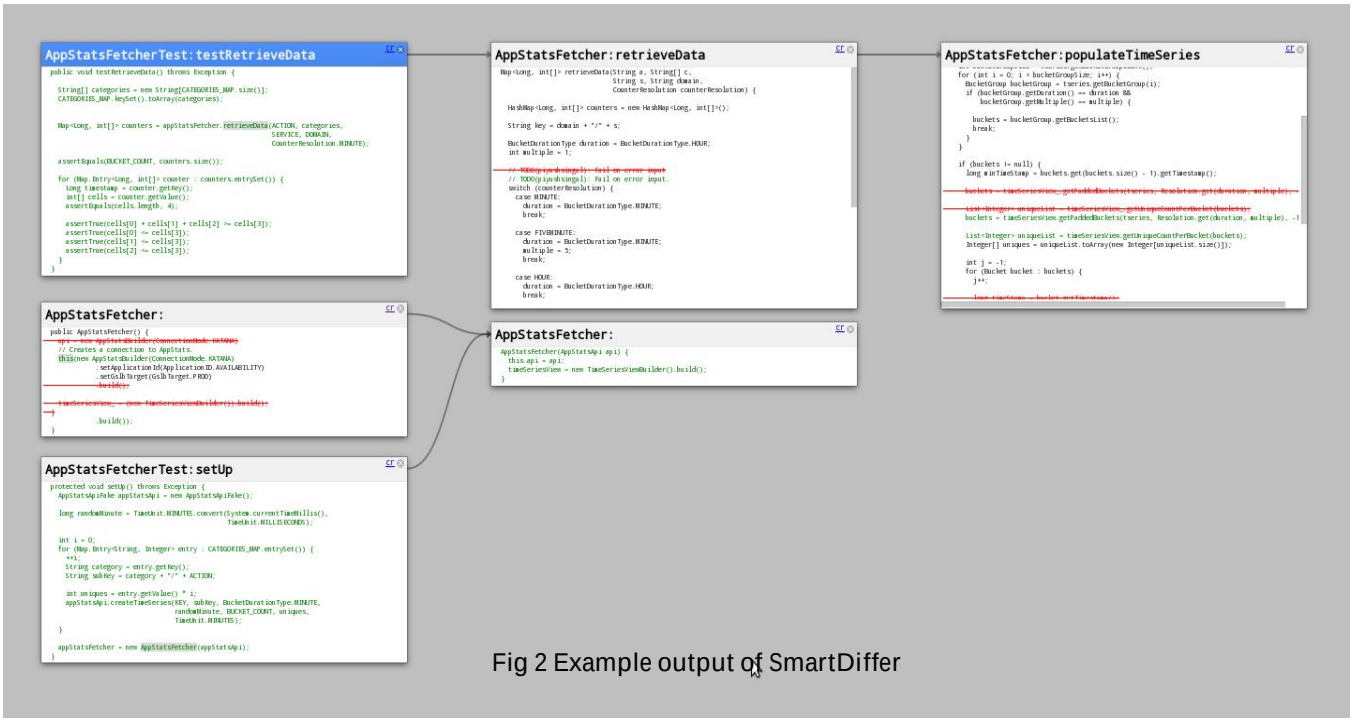

Fig 2 Example output of SmartDiffer

- In the old versions or deleted lines are highlighted with a red font and strikeout.
- In the new versions or added lines are highlighted with a bold green font
- Other lines are left as it is.

In order to read the method correctly, inter lacing of lines are done in such a way that old versions group of lines (constituting a continues changed lines) are placed before the corresponding new lines.

1. Merging old and new methods are done directly if the signature has not changed. However if the signature of the method has changed then mapping the old and new method is done using the following heuristics. If the name of the method is same but parameters / return types are different then the two methods are considered for in-line diff.
2. If method is renamed then we need to find the best match from the list of old methods. Best match is computed based on various weighted parameters:

    - Number of matching lines,
    - Percentage of changed lines
    - Size of the methods
    - Location of the methods
    - Presence of overloaded functions

The associations are used as edges in the call-graph and annotated with new / old / both. The nodes and associations are recorded in a persistent storage.

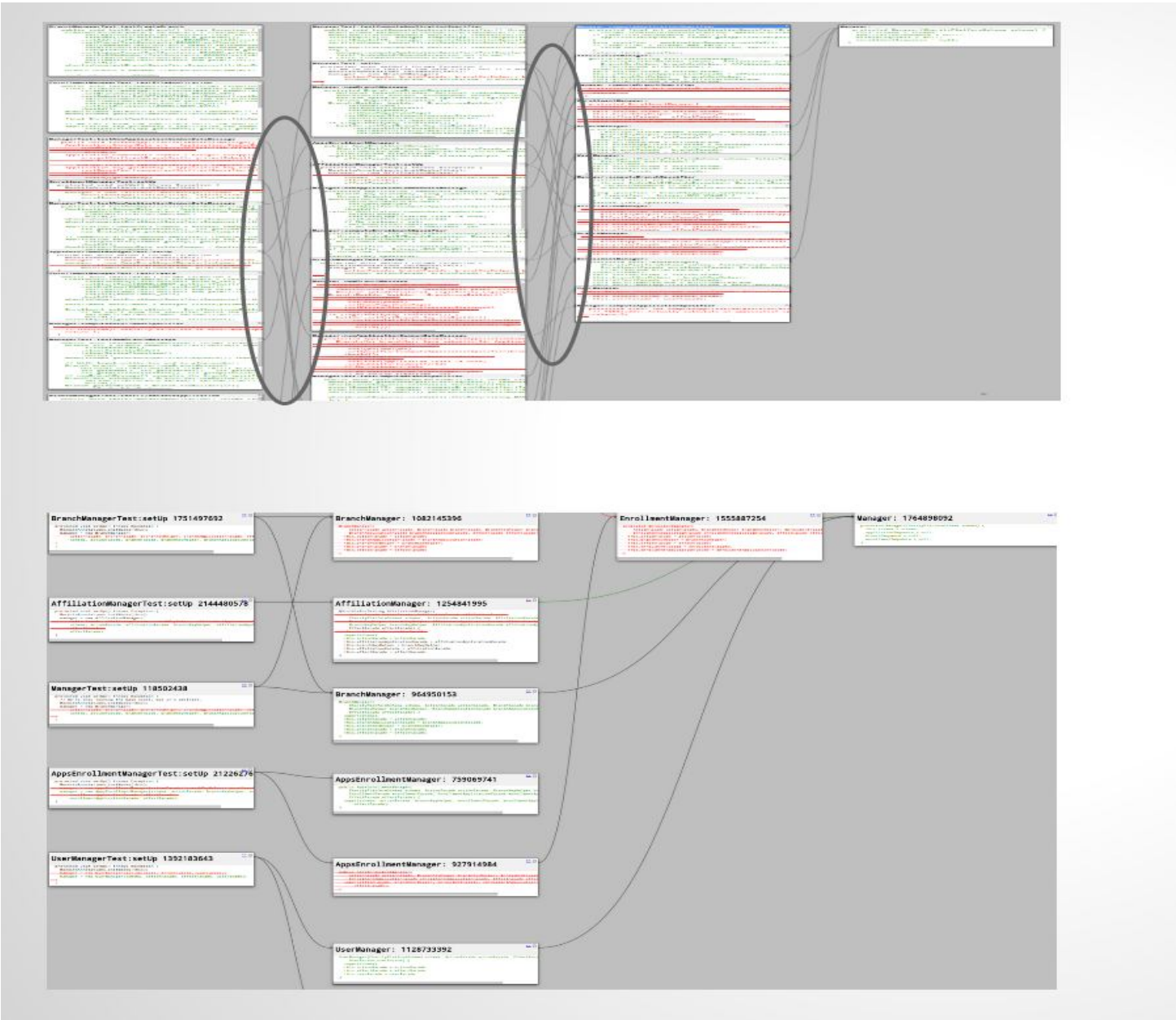

Fig. 3. Smart Differ Output before and after barycenter method of minimizing crossing

4 Architecture of Frontend

When the reviewer requests for the change-list the front-end server consults the layout engine for placement of the nodes [5] and returns diff graph for display.

4.1 Layout Engine

The Front-end talks to the Layout Engine, which computes topological order of the nodes to show caller-callee relationship. Barycentric [6] method is used to rearrange the nodes to get minimal edge crossings. Barycenter method was chosen as it works well for nodes that are fixed in a layer.

4.2 Client Display

The client renders the graph as a bubble view (fig. 2) with changed methods and callee-caller associations between the nodes with proper color. It also handles the other user actions for navigating, zooming and panning of the graph. It also has short cuts for reviewing only old code, only new code, or inlineed diff graph. It can expend the unchanged code by talking to the code repository. This gives the other context to the reviewer.

5 Results

This tool was used experimented with 100 change-lists. We observed that the change-lists with delta lines in the range of 50 to 200 lines were better reviewed using Smart Differ than simple file diff (fig. 3). Refactoring change-lists were also better viewed in smart-differ. Bigger change-lists cluttered the visualization with too many nodes. However by showing them as many sub-graphs separately and with better keyboard shortcut navigations, reviewers' experience can be improved for bigger change-lists as well.

6 Conclusion

This new techniques can be extended for any of the programming languages and will certainly help the author as well as the reviewer for quicker understanding of the changes. With keyboard shortcuts and mouse gestures, reviewers can get more context of the code under review easily. It is also possible to extend the caller-callee associations to any other associations and can be used to ignore the variable's name changes as well. Presenting the semantic information graphically, providing easy navigation and in-line diffs significantly improves the code-review process.

- *Arul Siva Murugan Velayutham was working at Google India, Bangalore. E-mail: arulsmv@gmail.com*
- *Ramakrishna Rajanna is currently working at Google India, Bangalore. E-mail: ramakrishna.r@gmail.com*
- *Devesh Yamparala was an intern in Google India, Bangalore. Email: dev344@gmail.com*

7 References

[1] D. S. Hirschberg, 'A Linear Space Algorithm for Computing Maximal Common Subsequences,' CACM 18 (1975) 341-3.

[2] J.W. Hunt and M.D. McIlroy An Algorithm for Differential File Comparison

[3] Goops - Googles Publisher Subscriber Techinique: http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html

[4] Ctags -EXUBERANT CTAGS : http://ctags.sourceforge.net

[5] John Warfield. Crossing Theory and Hierarchy Mapping. IEEE Transactions on Systems, Man, and Cybernetics, SMC-7(7):505–523, July 1977.

[6] Ismaeel, Alaa A. K.; Kumar Shukla, Pradyumn; Schmeck, Hartmut Efficient barycenter algorithm for drawing hierarchical graphs with minimum edge crossings