# Compare and Contrast: Visual Exploration of Source Code Examples

Rylan Cottrell[1], Brina Goyette[2], Reid Holmes[3], Robert J. Walker[1], Jörg Denzinger[1]

[1]Dept. of Computer Science
University of Calgary
Calgary, AB, Canada
cottrell, rwalker,
denzinge@cpsc.ucalgary.ca

[2]Robotics Institute
Carnegie Mellon University
Pittsburgh, PA, USA
bgoyette@cmu.edu

[3]Dept. of Computer Science
and Engineering
University of Washington
Seattle, WA, USA
rtholmes@cs.washington.edu

## Abstract

*Understanding the commonalities and differences of a set of source code examples can help developers to understand or to evolve application programming interfaces (APIs). While several approaches exist to assist developers in locating source code examples, they often present their results only in a basic list view, with at most an indication of the relationship to the search query; unfortunately, they offer no information on how the results relate to one another. A developer is then faced with the highly manual task of exploring these examples to discern their similarities and differences. This paper describes our prototype tool (called Guido) for exploring source code examples, using their structural correspondences. The Guido tool uses multiple coordinated views to visualize the relationships between examples, in order to assist the developer in identifying common and unique traits between them.*

## 1. Introduction

Software developers rely upon application programming interfaces (APIs) to leverage existing functionality written by others. To understand how to "best" use an API within their own systems, developers often examine other implementations—examples—that use that API. Conversely, for developers who create and maintain APIs, the need to evolve their public APIs must be balanced against the potential impact on the consumers of those APIs; by analyzing example usages of their API, the framework developer can gain an understanding of how other developers are using it. Unfortunately, analyzing even a small set of examples manually is a laborious and error-prone process.

Previous research has focused on locating source code examples (e.g., [3, 4, 7]) and presents this information in list-based views. Though such a list view can be helpful for certain tasks, it places a large burden on developers trying to infer commonalities and differences in the returned examples. Tools that identify the commonalities and differences between source code fragments (e.g., [5, 1]) are limited to pairwise comparisons; thus, simply sequencing the use of the location and comparison tools will not eliminate the large manual burden a developer faces in trying to explore a set of examples.

Current tools have two main shortcomings for these tasks: developers must keep track of where in the list they are and how the different examples relate through their commonalities and differences. To overcome these shortcomings, a technique should provide *locality*, *structure*, and *distinction*:

- **Locality**. During the exploration, the developer knows where they are.
- **Structure**. Structural information is not occluded from the developer.
- **Distinction**. The developer is able to identify commonalities and differences between the source code implementations.

We present our prototype tool, called Guido,[1] for the exploration of source code examples. Guido employs the Jigsaw framework [2] to create a generalization hierarchy from a set of source code examples. Utilizing this hierarchy, Guido replaces the traditional list view with multiple coordinated views [9] that support locality, structure, and distinction. These views are dynamic, updating in response to the developer's interaction.

This paper is structured as follows. Section 2 describes the related work and how it does not adequately address this problem. Section 3 describes the Guido prototype tool. Section 4 discusses remaining issues and future work.

[1]Italian, meaning "I guide"

## 2. Related Work

Developers wishing to explore software systems want to discover key artifacts and relationships that exist between the artifacts. Software exploration tools (e.g., RMTool [8], SHriMP [11], SEXTANT [10]) facilitate this discovery process by visualizing the different relationships between source code artifacts spread across a software system (e.g., type hierarchies, method callee relationships, dependencies). The problem of exploring interrelated examples is different from the general exploration of a software system, as the developer is interested in comparing and contrasting the examples.

Some clone detection techniques (e.g., CCFinder [6]) include a visualization element. Clone detection is quite different from analyzing commonalities and differences between examples, though one could imagine analyzing clones similarly for the sake of refactoring them [1].

A variety of tools (e.g., Semantic Diff [5], Breakaway [1], the comparison editor of the Eclipse IDE) can help assist developers to discover commonalities and differences between two source code examples. This limitation to pairwise comparisons hinders the developer's ability to explore larger sets of implementations. A developer using the pairwise comparison is still required to form a mental abstraction over the set as a whole.

## 3. Guido

We briefly outline the generalization graph that Guido computes, as well as its visualization and interactive features.

**Generalization graph.** Guido performs generalization from examples in a pairwise fashion, via the process described in detail previously [2]. This process maintains structural elements that each pair of examples have in common and elides the differences. For example, if we have two methods `m1` and `m2` (see Figure 1), we create a generalized structure by keeping the substructures the two methods have in common (visibility, return type, parameter type, and method call) and create *generalization variables* for those substructures that are in disagreement (method name, formal parameter name, referenced name).

Guido creates a generalization graph by performing pairwise generalization for all the examples; each generalization is added as a node to a graph with its originating examples as leaves. Identical generalizations are collapsed into a single node. This process is repeated on the generalizations themselves until a unique root node is created (such a root node always exists after a finite number of steps).

**Visualization.** Guido uses three coordinated views [9] to incorporate the three properties—locality, structure, and distinction—into a visualization: (1) the *hierarchy view* visualizes locality for the developer; (2) the *source view* visualizes the structural information and assists in visualizing distinction; and (3) the *tag cloud* visualizes distinction. These coordinated views can be seen in each of the subfigures of Figure 2. The visualization's initial state (Figure 2a) has the root generalization node selected.

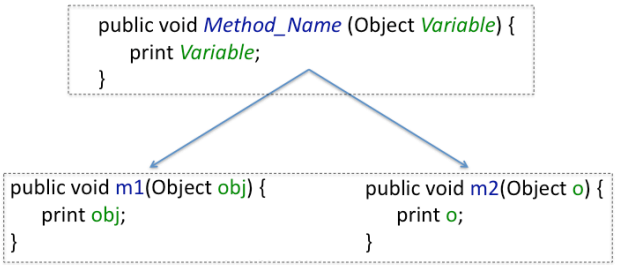


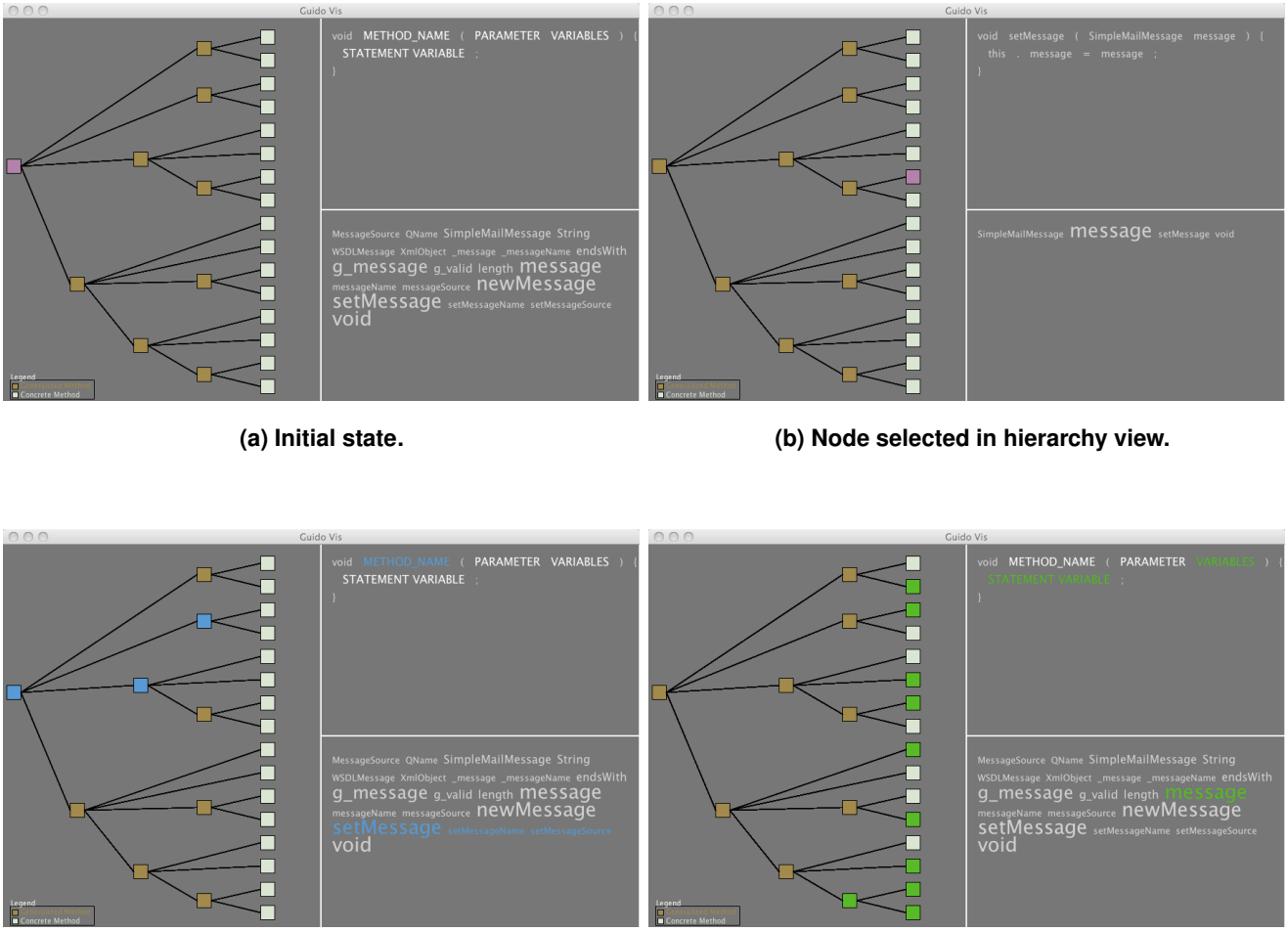

**Figure 1. Generalization structure (at top) formed from two concrete methods (at bottom).**

***Hierarchy View.*** The hierarchy view visualizes the generalization graph as a tree, where nodes with multiple parents are split into copies, one per parent; the root node is leftmost. (We discuss our rationale for presenting the graph as a tree in Section 4.) Generalizations are represented as brown nodes in the tree; off-white nodes represent the concrete implementations. A selected node changes to purple (Figure 2b). A right mouse click on a node will filter the tree view to only show the selected node and its children.

The tree layout manager places nodes horizontally first on the basis of their path depth from the root; the nodes at a given path depth are spaced uniformly from those at other depths to fill the view's horizontal dimension. Concrete nodes (i.e., leaf nodes) are spaced uniformly to fill the view's vertical dimension. The other nodes are placed by recursively traversing the tree from the leaves to the root; for non-leaf nodes, vertical position is set to the mid-point of its children.

***Source View.*** The source code view visualizes a synthetic textual representation of the source code present in any selected node; generalization variables are displayed in a larger font size (e.g., `PARAMETER VARIABLES` and `STATEMENT VARIABLE` in Figure 2a) and the commonalities are displayed at the regular font size (e.g., `void` result type in Figure 2a). To generate the block of source code, filters are applied to remove irrelevant information (from the perspective of comparing and contrasting examples) such as visibility modifiers and comments as they can cause screen clutter.

***Tag Cloud.*** The tag cloud represents the frequency of names within the nodes, considering types, local variables, formal parameters, fields, and methods (generalization variables are ignored). Using the maximum and minimum name frequency and the number of font sizes to be used (6 by default), we create a font size lookup table using a linear




**(a) Initial state.** **(b) Node selected in hierarchy view.**

**(c) Substitution variable selected in Source View.** **(d) Name selected in Tag Cloud.**

**Figure 2. Guido interactive effects.**

distribution model; more frequent names are displayed in larger font. This allows a developer to quickly see what are the most common names within the tag cloud as well as the more unique names. For example, in Figure 2a we can see the most common names (e.g., `newMessage`, `setMessage`, etc.) but we can also identify more curious, infrequent names (e.g., `XMLObject`, `WSDLMessage`).

**Interaction.** The interactions with the visualization have been designed to be simple so as to encourage exploration and discovery. An action in one view results in the other two views being updated. In order to help a developer understand the connections between the views, each view has a unique associated colour to represent selections made from it: purple for the hierarchy view, blue for the source view, and green for the tag cloud, as shown in Figure 2.

A selection of a node in the hierarchy view updates the source view with the generalization and the tag cloud with the selected generalization node (Figure 2b). Selecting a generalization variable or source code element in the source view will highlight the tags and nodes associated with it. For example, selecting `METHOD_NAME` in the source view of the initial state (Figure 2a) will result in three tags (`setMessage`, `setMessageName`, and `setMessageSource`) and three generalized nodes being highlighted (Figure 2c). The selection of a name from the tag cloud will highlight the associated source code elements or generalization variables in the source view and the nodes that contain that name in the hierarchy view. For example, by selecting the name `message` from the tag cloud, the generalization variables `VARIABLE` and `STATEMENT VARIABLE` plus 7 concrete nodes and 1 generalization node in the hierarchy view are highlighted (Figure 2d).

## 4. Discussion

We examine a number of remaining issues with the approach.

**Tree vs. DAG.** Our early attempts at visualizing the generalization graph used an unfiltered, directed acyclic graph. Unfortunately, these graphs tend to be non-planar, and interacting with them involves traversing paths from the root that can quickly become hard to follow; it is not obvious that 3D visualizations would completely mitigate this. We opted instead to create a tree structure as described in Section 3, as it eliminates those problems at the cost of inflating the apparent number of unique nodes. Determining which configuration represents the best choice for example exploration tasks will require further study.

**Scalability and performance.** The most significant impediments to scaling the technique lie in the cost of the generalization process, and the availability of screen real estate.

For the first level of generalizations, $O(n^2)$ pairs of examples must be generalized; due to the collapsing function, iteration quickly converges to a unique root node, but each generalization can be relatively expensive when examples are only somewhat similar (closely similar and dissimilar examples are fast to generalize). We have not yet undertaken a careful analysis of the asymptotic complexity of the process or of the bounds on the size of the resulting graph, but in practice, we have found that computing the hierarchy for (e.g.) 15 real examples can require about a minute. Screen real estate becomes a more serious issue more quickly: with the tree representation, a small number of examples can create a wide tree that does not readily display in the space available.

In combination with general usability improvements in future versions of the tool, we will consider visualization techniques like zooming and fisheye views for their compatibility with the needs of example exploration tasks, to deal with screen real estate issues. In addition, we are investigating a pre-processing step that (a) clusters examples on the basis of coarse-grained similarity, (b) computes fine-grained similarity only within clusters, and then (c) generalizes the clusters' generalized forms; this ought to both speed the process and eliminate edges that are less likely to be traversed during interaction.

**Implementation and integration.** As the tool was intended as a proof-of-concept, the cost of implementing its user interface was kept as low as possible. The visualization relies heavily on the Processing environment[2] which possesses significant, low-level conflicts with Eclipse. As a result, we present the visualization in a separate window from the Eclipse workbench. It is not clear whether a reasonable Eclipse perspective could be defined that allowed interaction with the visualization but without obscuring other important information needed by the developer. For example, there are cases where the developer wants to investigate the raw example source code in its original context, and possibly simultaneously view their own source code. Screen real estate has the potential to be a non-trivial issue here as well, though not an insurmountable one.

[2] http://processing.org

## 5. Conclusion

We have presented Guido: a visual approach that helps developers to identify commonalities and differences in sets of source code examples. Guido replaces the traditional list view with multiple coordinated views that provide the properties of locality, structure, and distinction for the examples through the use of structural correspondence. Future versions of the tool will address some of the issues that we have identified, at which point, formal evaluations will be undertaken.

## References


[1] R. Cottrell, J. J. C. Chang, R. J. Walker, and J. Denzinger. Determining detailed structural correspondence for generalization tasks. In *Proc. Joint Europ. Softw. Eng. Conf. and ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 165–174, 2007.

[2] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 214–225, 2008.

[3] S. Henninger. An evolutionary approach to constructing effective software reuse repositories. *ACM Trans. Softw. Eng. Method.*, 6(2):111–140, 1997.

[4] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, 2006.

[5] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proc. Int'l Conf. Softw. Maintenance*, pp. 243–252, 1994.

[6] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[7] A. Michail. CodeWeb: Data mining library reuse patterns. In *Proc. Int'l Conf. Softw. Eng.*, pp. 827–828, 2001.

[8] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *IEEE Trans. Softw. Eng.*, 27(4):364–384, 2001.

[9] J. Roberts. State of the art: Coordinated & multiple views in exploratory visualization. In *Proc. Int'l Conf. Coordin. Mult. Views Explor. Vis.*, pp. 61–71, 2007.

[10] T. Schafer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT software exploration tool. *IEEE Trans. Softw. Eng.*, 32(9):753–768, 2006.

[11] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Müller. On integrating visualization techniques for effective software exploration. In *Proc. IEEE Symp. Info. Vis.*, pp. 38–45, 1997.