

SDAT a Visual Source Comparison Tool

Rolf Biehn

University of British Columbia

ABSTRACT

This paper discusses a new technique to compare software programs and a new pixel map widget. By using an Abstract Syntax Tree to structurally compare two files or programs, many advantages can be realized such as better visualization, multiple language support and a reduction in the noise (such as a renamed method or variable). This paper discusses some of the issues of implementation of this approach and some of the solutions so far. In this paper I introduce a new widget for scalable and fully interactive pixel maps, called the "Mini-Map Scroller" that also supports orthogonal zooming and demonstrate how it can be used in the context of Source Comparison Utility.

KEYWORDS: Software Comparison, Pixel Map

1 INTRODUCTION

Software developers are often faced with the task of comparing two or more software versions. Typical usages of software comparison utilities include: A code-review prior to check-in, tracking down a recently introduced regression, and searching for code-clones in the source code (for future refactoring). SDAT ("The Source Difference Analyzing Tool") aims to assist users with comparing two separate, but similar programs together or two similar source files together.

SDAT will use an Abstract Syntax Tree (AST) and compare the input sources structurally. Many advantages can be realized over traditional comparison utilities by doing this. Some of the advantages include:

- Reduced noise (for example, renamed variables or re-ordered methods can be safely ignored during comparison)
- Cross-language comparisons become possible (such as C# to JAVA -- for companies which support programs written in multiple languages.)
- Better visualizations (such as syntax highlighting, special visualizations for different types of AST nodes, an over-view of precisely which files have changed structurally, etc...)
- Better user interaction (such as searching for a method by name, "jumping" to another method from one method call statement, filtering unwanted information, etc..)

I have implemented an algorithm for transforming JAVA ML (an XML AST representation) to an in memory AST model. I have also completed a very basic comparison algorithm. In this paper I will briefly describe these steps, introduce the Mini-Map Scroller (A pixel map widget capable of orthogonal zooming and direct user interaction) and 2 component views of SDAT that

specialize in visualizing the difference between two given methods in AST form.

2 RELATED WORK

Numerous source comparison programs exist such as Beyond Compare, KDiff, and WinMerge. However, the comparison algorithms for these programs only consider simple heuristics such as ignoring comments and ignoring white space, and do not use an AST to compare the source files. Several programs for XML comparison exist such as HTMLDiff, Araxis Merge and Guiffy. However, these programs are not aware of the Source Code format and therefore may introduce extraneous results (such as comments or renamed variables). All of the utilities mentioned in this paragraph only allow for one file to be compared at a time, and lack easy inter-class navigation abilities.

The problem of source code comparison was briefly addressed by Chevalier et. al[4]. Munzner et. al[5], Bauman et. al.[6], and Holten and Wijk[7] explored issues regarding tree comparison visualizations. Several programs are available for visualizing source code such as Relief (which uses a radial gliff system) and aiCall (which uses flow-charts). Both Voinea et. al[8] and Jones et. al. [9] used pixel maps to indicate areas of interest in source code files. However, pixel one line cannot represent more than one document line and therefore multiple or large display maps are required in order to represent large sets of data. Appert and Fekete[10] introduced the idea of orthogonal zooming. I wish to extend this idea towards pixel-maps.

3 IMPLEMENTATION

3.1 Overview

The AST process can be divided into 5 distinct phases. The source files are first transformed into an AST model. After which, an analysis phase is conducted. In this phase, the mappings are determined automatically (for example, class X maps to class Y, method X maps to method Y) and then a comparison is conducted. These results are sent to a Visualization program. The next stage is user interaction; however this phase is out of scope for this project.

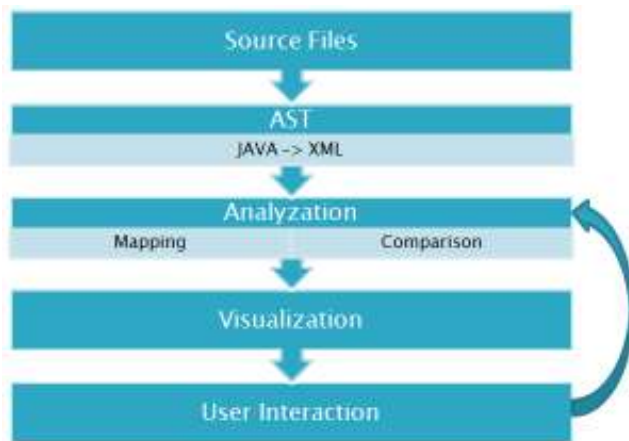


Figure 1 The 5 stages of AST Comparison

3.2 AST Phase

In the AST Phase, I used JAVA ML[1] to translate java source files into an XML model. This XML model is then translated to my own AST object model by using the JDOM[2] library. I chose two functions from 2 consecutive revisions in SWT's[11] Control class to be used as the data-set. The entire process is described in Figure 1

3.2.1 Discussion

Several Obstacles were discovered in this phase. JAVA ML does not support JAVA 5 and above nor does it store the symbol table information(required for the mapping sub-phase and the User Interaction later on). JAVA ML does not use an XML serializer; therefore it does not escape characters such as apostrophes and angled brackets. I resolved these issues by manual data manipulations. Several JAVA ML nodes have attributes such as "methodName", "className" & "variableName". I changed these attributes to "name" to support greater abstraction. I propose that JAVA ML adopts this standard in the next release.

I learned that the memory requirements for an XML representation are considerable. 5,000 lines of code require 800KB of memory. Of course this large memory consumption makes sense when I consider all of the duplication of node and attribute names text in the XML representation and also a textual representation of an integer will always require more bytes than its binary counter-part. My original design involved creating wrapper classes for each of these XML nodes and keeping the XML nodes in memory, however this has proven to be infeasible.

3.2.2 Conclusion

The ability to serialize the object model for debugging is desirable; however, XML as the primary object model is impractical. In future, I will use a binary representation of the AST Model as the primary output of this phase, with the ability to serialize / deserializer to XML for debugging purposes.

3.3 Comparison

In this phase meta-data is added to the AST node model to indicate differences and missing nodes. The process is described in more detail in Appendix A Figures A1 to A3.

3.3.1 Discussion

I originally attempted using several open source libraries for java for xml comparison, however I was unable to find a library that met my requirements and would be easy to learn in a relatively short period of time. Therefore, I wrote my own comparison algorithm it is very simplistic. It can only compare nodes that exist in the same block-level. The comparison algorithm will start with a node on the "left" tree. (lets call it L(1)). It will look at the corresponding statement to the right (call it R(1)). If L(1) and R(1) are an exact match, we move to the next statement on each side (L(2) and R(2)). If a node with the same AST type is found, it is considered to be a similar node. If one of the attributes or children are missing or incorrect, they are marked as such. The algorithm then continues to L(2) and R(2). If L(1)'s AST node type cannot be matched to R(2)'s node type, the algorithm will look ahead up to 20 lines (arbitrary number) on the right hand side in order to find the next exact matching AST node (in the same block). If an exact matching node, (call it RX) can be found then R(1)->R(X-1) are marked as missing. Afterwards, the algorithm continues at L(2) and R(X + 1). On the other hand, if it cannot be found on the right hand side, the left side is searched for an exact match to R(1) up to 20 lines. If a matching L(X) can be found, then L(1)->L(X-1) nodes are marked as missing and the algorithm continues at L(X + 1) and R(2). If neither of these conditions are met, L(1) and R(1) are both marked as missing and the algorithm continues at L(2) and R(2).

3.3.2 Conclusion

Clearly this approach is inadequate, however, time constraints prevented a better implementation. In future versions, I intend to investigate existing techniques for text and XML comparison as well as specific software program comparison heuristics and generalities in comparison.

3.4 Visualizations

A program can be thought of as a very strict hierarchy of nodes, each of which contains properties on every hierarchical level. For example the class-level hierarchy contains properties regarding member variables, class name, visibility, etc... I have simplified this hierarchy for visualization purposes because I do not think that most developers care about the level of detail after statement. (See Appendix A Figure A4: for a description of all the levels). The high level plan for SDAT is to create multiple views of this tree which specialize in viewing specific levels. However, for this project only the method level and below will be considered.

The visualization of this project is composed of 3 major components, pictured in (Appdenix A Figure A5). The Method View specializes in showing the differences of method using a tree widget, the Mini-Map Scroller is used as an over-view in this view and the Detailed View is useful for showing more detailed information for statement differences. The initial implementation has been programmed using JAVA Swing. JAVA was chosen because it has a flexible library for performing tree operations (known as a JTree in JAVA) and it is cross-platform.

3.4.1 Model Adaptation

Before any visualization can occur, the AST model must be adapted into something more suitable for a visualization for developers. As an example, the statement "mName(arg1)" has several children in the AST model. Clearly multiple child for

this statement is not a desirable visualization. In order to visualize a statement, each node supports an ability to flatten itself into a stream of tokens (also including tokens for meta data information). This is also the data that gets sent to the Detailed View.

Another example of adaptation occurs for the “if statement”. An if statement has 3 possible children in the AST model (a condition, a statement clause, and optionally, an else clause), but clearly it is desirable to have a model that resembles how a programmer would write this statement. (i.e. with the “if keyword”, the condition statement on the same line and the statement clause and else clause as children of this node).

3.4.1.1 Discussion

Unfortunately, using ASTs for visualization of a statement suffers from several drawbacks. For starters, the user’s original code formatting is not shown. (Ideally an option to allow the user to format the code how they please could be supported.) In addition, if the statement is extremely long-- spanning several lines-- it will be shown all on one line. Most troublingly; user comments are completely lost in this process. However, the AST representation may be more appropriate for comparing cross-language comparisons where the difference in language syntax may make it difficult for the user to recognize the differences. I will re-analyze this approach in the future and perhaps just show the original text of the statement from the original source file. The AST generation logic must be improved to maintain character indexes and comments in this case.

3.4.1 Mini-Map Scroller

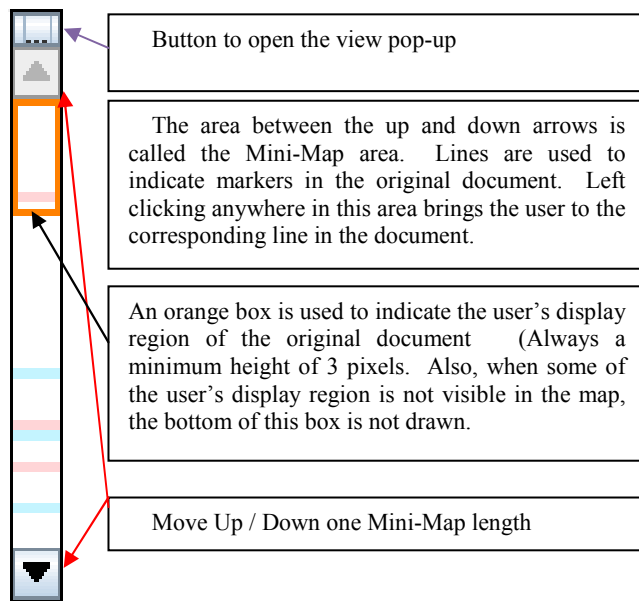


Figure 2 : Basic mini-map scroller

The Mini-Map Scroller is designed to work with any document with a concept of lines and markers. Markers are user objects consisting of a colour and a line number. It is recommended, but not required, that these colours match the same colours used in the document. In practice the Method View uses a Tree structure as a document. The required interface for a component to use the Mini-Map can be found in

Appendix A. The image at Figure 2 explains the basic interaction mechanics of the Mini-Map.

2.1.1.1 The Pop-up Viewer

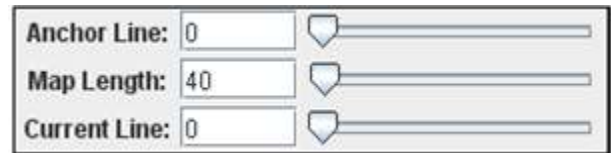


Figure 3 – the Pop-up Viewer

After clicking the view pop-up button the user is presented with the view pop-up box (pictured in Figure 3). This pop-up box will remain visible until the user clicks on the pop-up button again. (I will implement a close button in the next version). The pop-up view also serves as a simple visualization of the user’s current view via the sliders. The Anchor Line is the first line displayed in the Mini-Map. The map length refers to how many lines the Mini-Map should show at a time and the current line refers to the top line of the document view. All of these values can be inputted directly into the text boxes or via the sliders. I changed the slider UI to respond to a single click gesture (i.e. click on the knob, move the mouse left to right, and then click again to commit the value). I hope that this new gesture makes it easier on the user to select certain areas because no drag clicking is required. I postulate that drag clicking, makes it more difficult to move the mouse due to the additional muscle control required. The anchor line will update automatically if the Mini-Map determines that it cannot fit the current line into the current map view. The layout of the pop-up widget is east to west. I considered laying out the sliders north to south (for a more intuitive user experience), however this approach would cause a much larger number of lines to be obscured by this pop-up and such a layout may be counter-intuitive when using the orthogonal zooming feature.

3.4.2.1 Mini-Map Scalability – Display Region

The Mini-Map Scroller allows for display scalability by using several different techniques. It is able to give an overview for an arbitrary size of document. If the user specifies a map length that is currently less than available height for the Mini-Map area then it is clear that we can visualize the markers by lines with a height of $\text{floor}(\text{map size} / (\text{Mini-Map height}))$. When the map size exceeds the height of the Mini-Map height, a percentage of the width is used to represent a region in the original document. In this case, the number of occurrences of a marker colour in a region are counted and the percentage of the total of the number of lines in the region determines how much width is allocated to that colour. Each unique marker colour is guaranteed a minimum of one pixel width, while the remaining width is divided up evenly, according to the percentages, between markers and non markers. Obviously it is only necessary to traverse the current markers to determine this information, making this algorithm run in $O(M)$ where M is the number of markers.

The greater the width, the more expressive the line can be. Conversely, as the width becomes smaller, expressive capabilities are lost. Care should be taken by the developer to not encounter a situation in which the number of possible

markers exceeds the width of the Mini-Map. It is entirely possible to allow the user to dynamically resize the Mini-Map as no calculations are done using hard-coded values.

When the height is less than the map size, the scroller (i.e. the orange box) will be at its minimum height of 3 pixels. This would be rather difficult to click and drag using the traditional scroll-bar interface. Therefore, a new interaction mechanism has been introduced. The user is able to engage a “locked mode” by right clicking anywhere in the Mini-Map area. After right clicking, the user’s mouse pointer is changed to a hand and immediately moved to the top of the scroller. In this mode, the user is prevented from moving left to right, but they can still move up and down and click on the up and down arrows. While the user moves the mouse, the current view (and slider) is updated. Rotating the mouse wheel up and down will move to the previous and next marker respectively. Left clicking in the scroll area will return the user to normal mode. If the user right clicks while in the locked mode, the user is transformed into “resize mode” (indicated by the mouse cursor changing to the east-to-west resize cursor). While in resize mode, the user can move the mouse left to right to change the map length. However, this interaction performs as a linear scale making it quite tedious to change size of the map. Right clicking again from the resize mode returns the user to locked mode. Left clicking in the Mini-Map scroll area from the resize mode returns the user to the normal mode. (The locked and resize modes are also available when the map size is less than the Mini-Map height).

3.4.2.2 Mini-Map Scalability – Performance

# of Nodes	# of Markers	Expand(ms)	Marker Calc.(ms)	Rendering Time (ms)
10,000	2383	243	16	< 50
50,000	11646	460	134	< 50
100,000	23,251	1912	499	< 50
200,000	46338	4554	2187	< 50

Figure 4 – Mini-Map performance results

Figure 4 illustrates the results from my stress-testing runs of the Mini-Map. The “Expand” column refers to the time it takes for a node to completely expand after clicking the expansion icon. This time includes the time it takes to re-calculate the markers and for Java to perform a JTree expansion. The “Marker Calc.” column is the time it takes to solely recalculate the lines for all of the markers and the “Rendering” column is the time it takes to render a Mini-Map that uses all of the lines once the markers have been calculated. The stress testing machine was a Dual Core 2.8GHz Athlon computer with 2 gigs of RAM running Windows Vista. The data was created at random and used in a single view of the Method View. Ten independent trials were run and the average of these trials was displayed.

The recalculation of markers appears to be the biggest bottle neck for performance. The underlying reason for this is every marker must be visited in order to update its line number, and in order to determine its new number, it must look itself up in the tree (for an average of $M \cdot \log(n)$ per recalculation where M is the number of markers and n is the number of nodes). Perceived performance could be increased by running the calculation in a

background thread once the JTree has collapsed (perhaps showing a calculating animation in the Mini-Map itself) It should be possible, under certain situations, to specify markers as blocks (i.e. lines 5 to 50 are blue) thereby reducing the number of marker re-calculations needed. It is also possible to make the Mini-Map Scroller only request the markers needed for the current rendering therefore potentially differing the need to re-calculate certain markers. . These approaches have not been implemented in the current version, but are possible in future versions.

3.4.2.3 Mini-Map Scroller Discussion

I asked a group of 3 of my computer programmer colleges to try the Mini-Map scroller. All 3 of them described it as being easy to use, once the interaction mechanisms were understood (although they were all unable to discover the resize & locked mode on their own). In future, to address this problem, I will experiment with a tool-tip like overlay on the Mini-Map that says “Try right clicking me”. Once the user interacts with the Mini-Map or after a set amount of time, this overlay will disappear. My colleges did not find the resize mode to be useful.

I believe every scrolling-widget should adopt the “locked mode” behaviour as it offers some advantages over the traditional scrolling widget such as being easier to click when the scroller is very small and less stress on the hand. The Mini-Map provides an adjustable over-view of all the changes. Future enhancements include further optimization, a check-box to specify that the user wishes to see the entire map at all times and a more rigorous SDK available for public consumption.

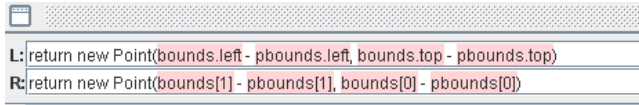
3.5 Method View

My original idea for method visualization was to use two side-by-side flow-chart representations of the AST model and colouring nodes that are different. However, I now believe this to be an incorrect approach for two reasons. #1) A statement bound visualization would be closer to a developer’s day-to-day mental model of the code and #2) this approach does not take advantage of position particularly well, which is one of the most import aspects of visualizations. Two views are placed side-by-side. Each view has a tree to visualize statements and a Mini-Map to explore the tree. Blocks of statements can be collapsed and expanded at will. Nodes that are different are highlighted in a soft red, missing nodes (also known as orphan nodes) are highlighted with a soft blue and nodes with collapsed children with markers are highlighted with purple. The colour scheme was verified as appropriate for colour-blind people by using VizCheck[3]. When a node is collapsed or expanded, all of the subsequent marker locations can change. It is important to not traverse the entire tree structure in this case. The algorithm avoids this by maintaining a list of all markers in the document (via the full tree path) and updating a cached version with the correct line numbers when appropriate. (It’s only necessary to update lines that are greater than or equal to the node being expanded or collapsed). The Method View also supports synchronized scrolling and synchronized selection.

3.5.1 Discussion

I’m not completely sold on the value of two Mini-Maps. One Mini-Map may be more appropriate. Also it may be more appropriate to have this map to the left instead of in the middle. (so it is less obtrusive),

3.6 Detailed View



```
L: return new Point(bounds.left - pbounds.left, bounds.top - pbounds.top);
R: return new Point(bounds[1] - pbounds[1], bounds[0] - pbounds[0]);
```

The Detailed View is used for a more detailed, line by line comparison. It is important to support this view because some of the statements may exceed 80 characters and will be very difficult to see in the non-full width Method View pane. The Detailed View reacts to the active selection in the Method View via a SelectionService. (I have implemented that myself, similar to Eclipse's selection mechanisms.) The input to the Detailed View is a stream of tokens generated from the AST Model. The implication for this is that the highlighting will be slightly different than a pure textual comparison. For example, a textual comparison is likely to regard the "bounds.left" text and "bounds[1]" as being different only after the s of bounds, but an AST comparison considers these different because an index reference is a different type of AST node than a regular variable reference (although this behaviour could be changed).

The long term plan for this view is that it provides detail for any active view and active selection.

3.7 Visualization Limitations

Currently the only way to invoke the views is via command line arguments. In the future, I hope to provide an outline tree widget view which contains the hierarchy of the packages and classes. The user will be able to use this outline view to drill further into the differences. Also I am using JDesktops so the user is free to resize anything they like. In future, I'd like to implement this as an Eclipse RCP app to allow pre-defined views and a better work-bench like experience for the user.

4 USER SCENARIOS

A regression has been discovered and assigned to a developer. The current source code is compared to a previous version where the regression does not exist. The developer is familiar with program so uses an outline view to target suspect areas of changes. The method view is used for more details on differences in the methods and the problematic check-in is discovered.

A user wishes to review his changes before checking them into source control; therefore they wish to see all of the differences prior to checking in the code. The user navigates every difference by using the next difference navigation, confirming the changes made.

5 CONCLUSION

I have discussed and implemented a method to visualize differences for methods from an AST comparison. I suggested several different areas for further investigation and improvement. I have created a new widget capable of displaying pixel maps at arbitrary, user controlled resolutions. At the current time, my system does not fully utilize the power of the AST Model, however a lot of potential exists with this

approach and the first required stages for this have now been completed.

6 FUTURE WORK

Many areas that are needed to create a complete software comparison program have been left untouched in this project. In the future, we must support more AST nodes from multiple languages, a better, more rigorous comparison algorithms, the ability to compare and visualize classes, packages, etc.; and of course more user interaction (such as filters, go to, multiple views, a navigation pane and navigation history, changing the link mapping, etc...) Some interesting tangents for further investigation include: Creating the AST from a binary, other applications for the Mini-Map Scroller and user studies.

7

7 REFERENCES

- [1] Greg J. Badros. JAVA ML : An XML-based Source Code Representation for Java Programs. [Online] <http://www.badros.com/greg/JavaML/>.
- [2] JDOM: <http://www.jdom.org/>
- [3] VizCheck: <http://www.vischeck.com/>
- [4] Fanny Chevalier, David Auber, Alexandru Telea. Structural Analysis and Visualization of C++ Code Evolution using Syntax Trees. *9th International Workshop on Principles of Software Evolution (IWPE 2007)*.
- [5] Tamara Munzner, Francois Guimbretiere, Serdar Tasiran, Li Zhang, Yunhong Zhou. TreeJuxtaposer: Scalable Visibility. *SIGGRAPH 2003*, published as ACM Transactions on Graphics 22(3), pages 453--462, 2003.
- [6] Shannon Bauman, James Clawson, Josh Cothran, Jeanie Miskelly, Zach Pousman. Treefoil: Visualization and Pair Wise Comparison of File System Trees.
- [7] Danny Holten and Jarke J. van Wijk. Visual Comparison of Hierarchically Organized Data. *10th Eurographics/IEEE-VGTC Symposium on Visualization (Computer Graphics Forum; Proceedings of EuroVis 2008)*, Pages 759 - 766, 2008.
- [8] Lucian Voinea, Alex Telea and Jarke J. van Wijk. Evolution, CVSscan: Visualization of Code. *Proceedings of the ACM 2005 Symposium on Software Visualization*, St. Louis, Missouri, USA, May 14-15, 2005 2005.
- [9] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization for Fault Localization. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference* pages 54- 56.
- [10] Catherine Appert and Jean-Daniel Fekete. OrthoZoom Scroller: 1D Multi-Scale Navigation. *Proc. SIGCHI 06*, pp 21-30.
- [11] SWT: The Standard Widget Kit: <http://www.eclipse.org/swt/>

APPENDIX A

```
public interface IMarker {

    /**
     * @return the Color of the Marker
     */
    public Color getMarkerColor();

    /**
     * @return the Line number of the marker
     */
    public int getLine();
}

public interface IMinimapAdapter {

    /**
     * @return a list of all the markers in the document. These markers must be
     *         ascending in sorted order.
     */
    public List<IMarker> getMarkers();

    /**
     * Ensures that lineNumber is visible
     *
     * @param lineNumber
     */
    public void goTo(int lineNumber);

    /**
     * Returns the total number of lines in the documents
     */
    public int getTotalLines();

    /**
     * @return the first visible line of the document
     */
    public int getFirstVisibleLine();

    /**
     * @return the last visible line of the document
     */
    public int getLastVisibleLine();

    /**
     * @return The JAVA component used to render the view of the document. This
     *         is typically a JViewport or a JScrollPane.
     */
    public Component getComponent();
}
```

```

public static void print() {
    System.out.println("Hello World!");
}

public static void print() {
    System.out.println("Hello Universe!");
}

```

Figure A1 – the original two inputs.

```

<method name="print" visibility="public" static="true">
  <type name="void" primitive="true"/>
  <formal-arguments/>
  <block>
    <functionCall="println">
      <target>
        <field-access field="out">
          <var-ref name="System"/>
        </field-access>
      </target>
      <arguments>
        <literal-string value=""Hello World!""/>
      </arguments>
    </functionCall>
  </block>
</method>

```

Figure A2 –The results after the JAVA ML AST generation

```

<method name="print" visibility="public" static="true">
  <type name="void" primitive="true"/>
  <formal-arguments/>
  <block>
    <functionCall="println">
      <target>
        <field-access field="out">
          <var-ref name="System"/>
        </field-access>
      </target>
      <arguments>
        <literal-string value=""Hello World!"">
          <meta value="diff"/>
        </literal-string value>
      </arguments>
    </functionCall>
  </block>
</method>

```

Figure A3 –The results after the injection of meta data.



Figure A4 –The simplified hierarchy of a program.

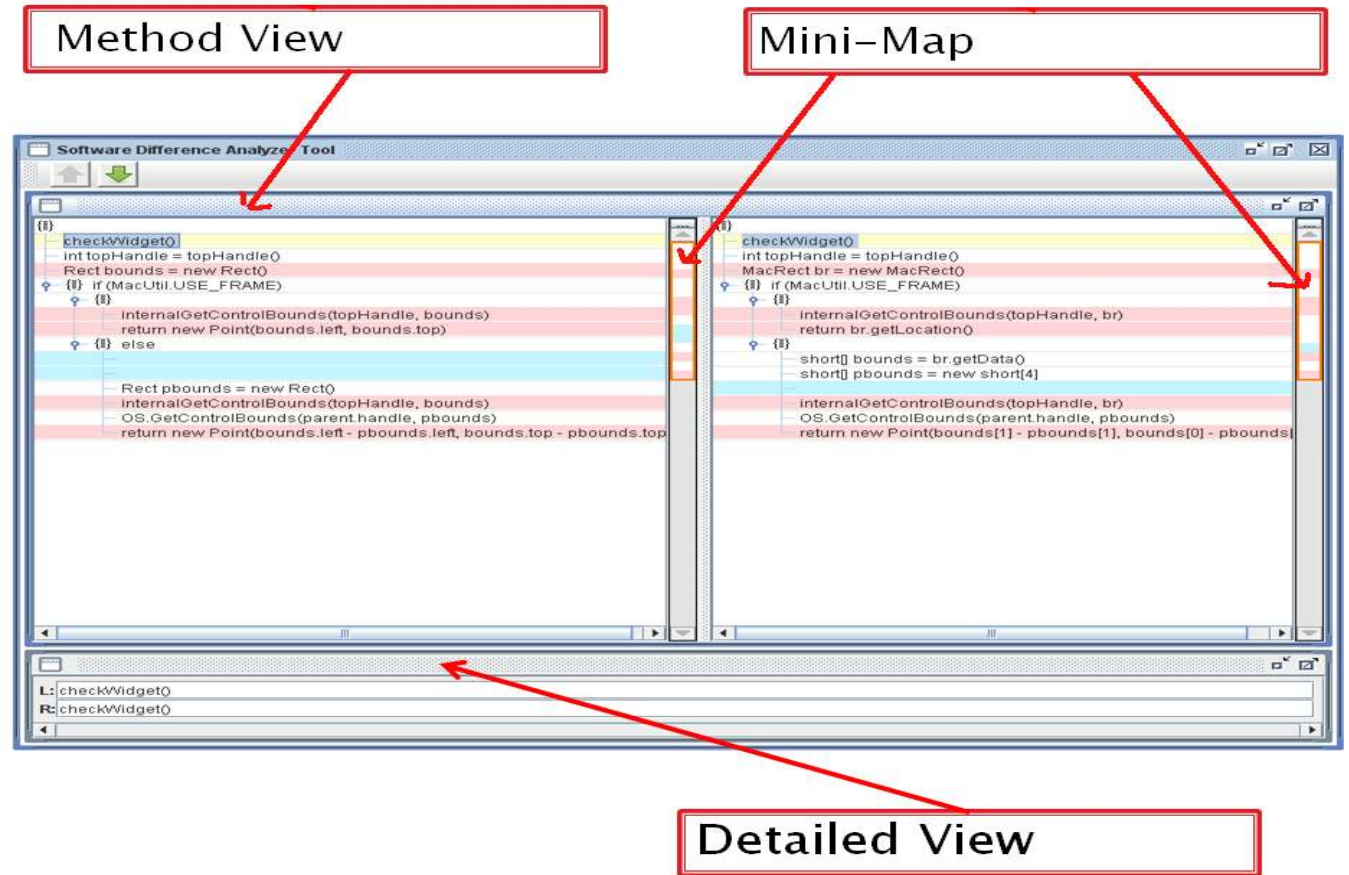


Figure A5 –The three major components of the Visualization