# Step 2 Report

Sunday, February 14, 16

Our general goal is to describe where logging calls happen in the source code using anti-unification. In the previous step, we investigated what Jigsaw does and how to use it to create a CAST (Correspondence AST) from a pair of Java classes that use logging calls. In this step, we aim to propose an algorithm to construct an anti-unifier from a CAST with special attention to the location of log statements. Our intention is to abstract common pieces from the two input structures.

As mentioned in the first report, each node in the CAST maintains a list of correspondence connections or elements, in which each of them represents an anti-unifier. However, we should construct one single anti-unifier that is helpful to solve our problem; thus, our algorithm should select an anti-unifier of the best fit to our application. In general, there is no unique optimal fit, but our desire is to find one of the best fits to approximate the optimal one that can sufficiently solve our problem.

The following steps could be taken to come up with an anti-unifier to abstract the structure of two input seeds containing logging statements:

1. Our algorithm gives ASTs of two Java classes that use logging calls, as a pair of input seeds, to the Jigsaw framework. Jigsaw generates a CAST (Correspondence AST) from both seeds. We create a list of correspondence connections from the CAST, in which each element has a similarity value above a pre-determined threshold.
2. We approximate optimal fit using a greedy algorithm that:
   1. Selects the best correspondence connection to our problem for each node in the CAST (If there is any).
   2. Remove all other correspondence connections involving those two nodes from the list of correspondence connections.

Overall similarity value between the two Java classes containing logging calls is computed by summing up Jigsaw similarity of all correspondence connections in the list and dividing it by the number of nodes in the largest input AST.

1. For each correspondence connection in the list generated from the previous step, two AUNodes should be created from nodes involving in that correspondence element. Each AUNode contains one ASTNode and its properties. Properties of a node are stored in a map data structure that associates each property name to its values. We construct an anti-unified AUNode (anti-unifier) from both AUNodes that abstracts their structures.

In the following, we try to illustrate our algorithm in more details using two examples of Java classes containing log statements shown in Figure 1 and Figure 2.

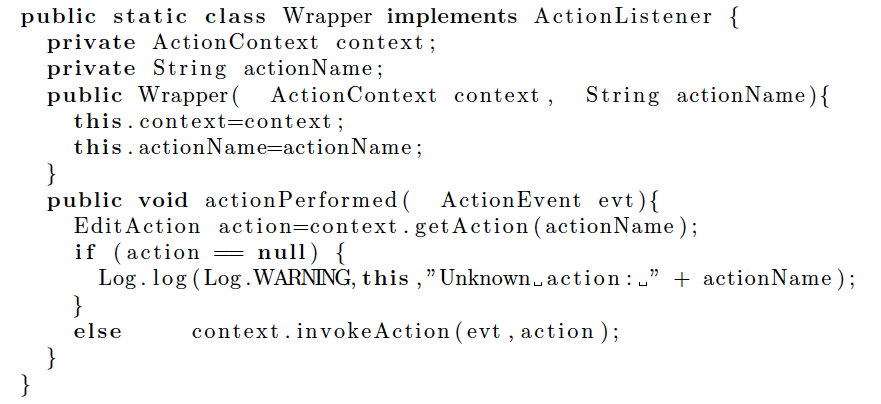


Figure 1: A logged Java class, Example 1

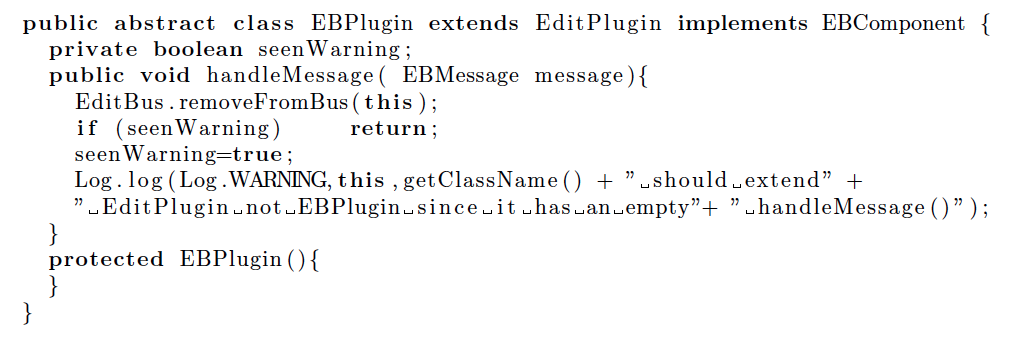
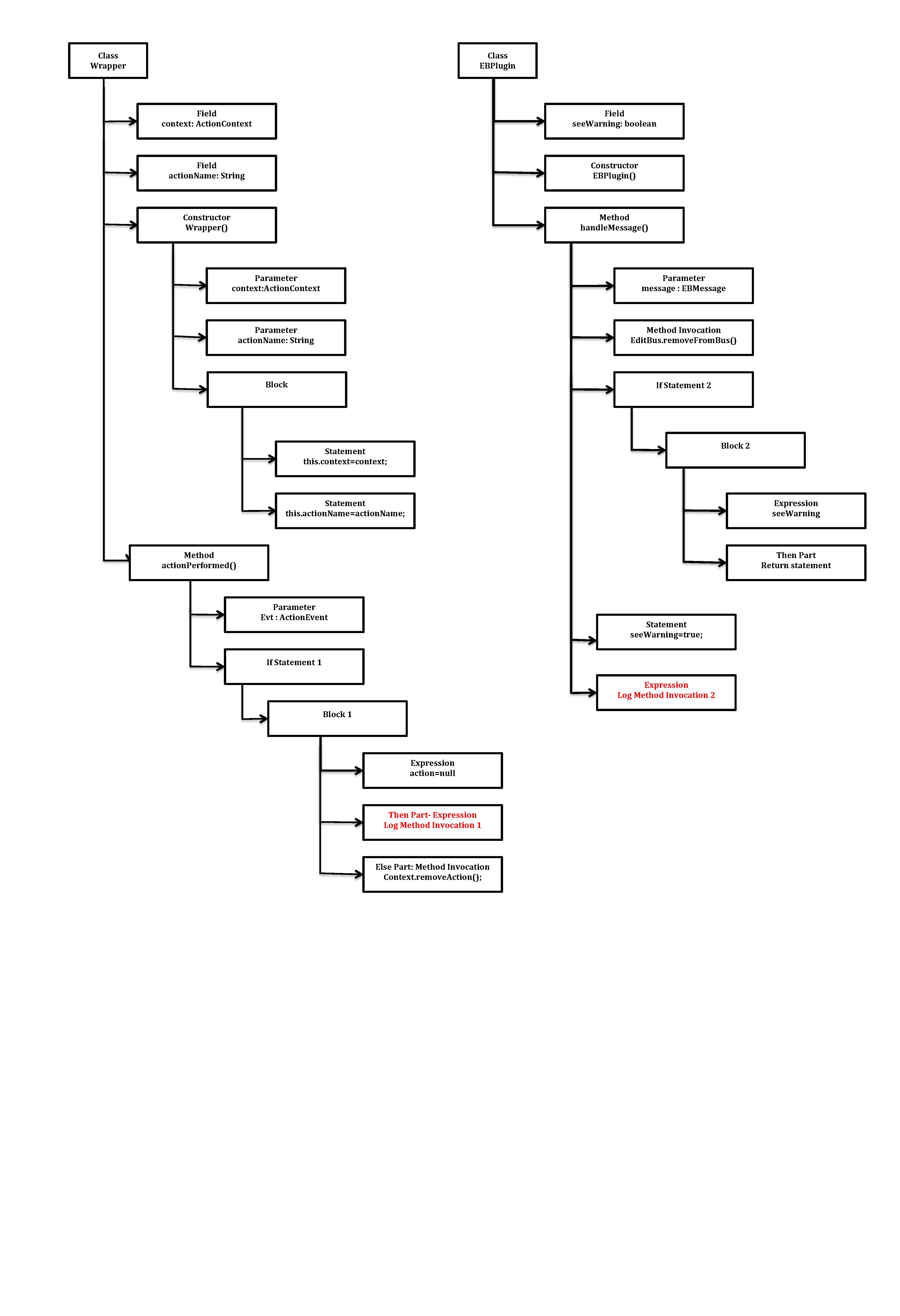


Figure 2: A logged Java class, Example 2

In the **first step**, our algorithm gives ASTs of the two Java classes, shown in Figure 3, to the Jigsaw framework as a pair of input seeds. Jigsaw generates a CAST from both seeds, in which each node holds a list of its correspondence elements; each correspondence element is a connection between two corresponding substructures from a pair of input ASTs along with a similarity value, which is computed by Jigsaw, to indicate the level of correspondence between these two substructures. A pre-determined threshold value[[1]](#footnote-1)is set for Jigsaw to remove all correspondence connections below a minimum similarity value. Algorithm 1 creates a list of all correspondence connections for the CAST generated from our examples with similarity value above the threshold. This algorithm sorts the list of correspondence connections in a descending order based on the similarity measurement. The generated list for our examples is shown in Figure 4.

Figure 3: ASTs of logged Java classes in Figure1 and Figure2

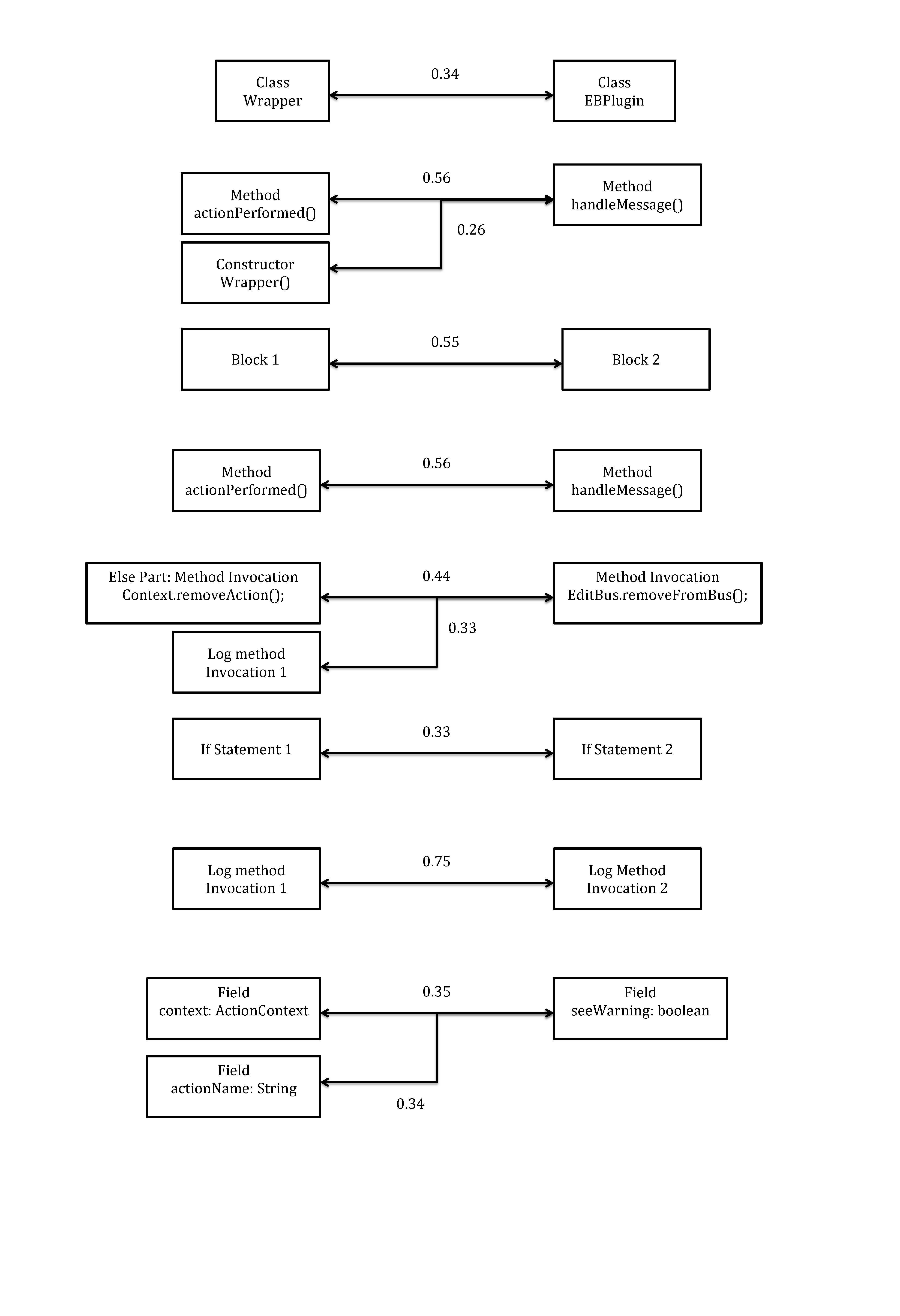
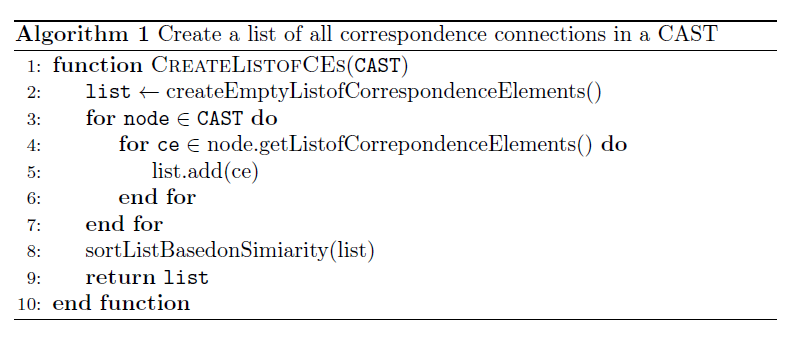
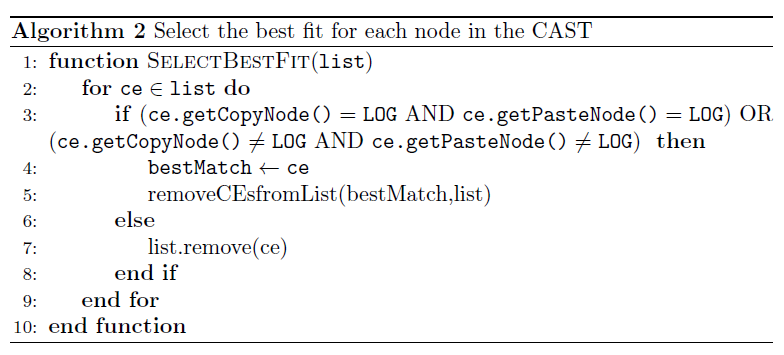
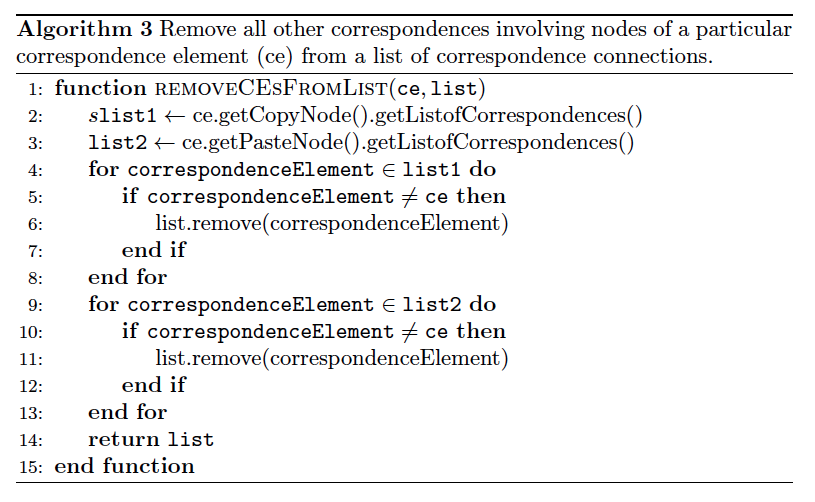


Figure 4: List of correspondence connections



In the **second step**, we aim to approximate optimal fit using a greedy algorithm by selecting the best correspondence to our problem for each node in the CAST (If there is any) and removing all other correspondences involving those two nodes from the correspondence connection list as described in Algorithm 2 and Algorithm 3. As a result, each node can either be anti-unified with its best fit in the other seed or nothing. According to Algorithm 2, correspondence connections for the nodes that could not be matched with another node will be removed from the list.





Algorithm 2 demonstrate how we select the best fit for each node based on the following assumptions and constraints:

1. Each log method invocation node should be matched with a corresponding log method invocation in another seed or nothing
2. The best correspondence have the highest similarity value compared to the other options

As an example, when two log method invocations matched with each other, the other correspondence connections involving these two nodes will be removed from the list of correspondences.

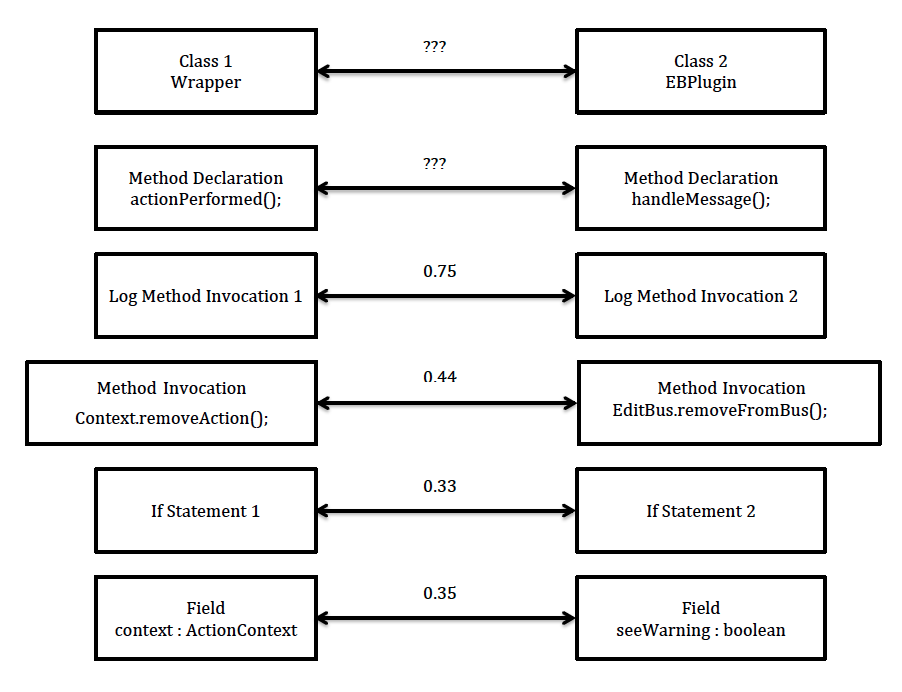


Figure 5: List of correspondence connections

For the other nodes we select the best correspondence connection with the highest similarity value and remove the other connections from the list. Figure 5 shows the list of correspondence connection for our examples. As you can see, two if statements are selected as the best match for each other; so, these two substructures should be anti-unified together meaning that their children nodes should be anti-unified with each other as well, while one of them contains a log statement and there is no corresponding log statement in the other substructure. That is contrary to our first assumption to anti-unify log statement with another log statement or nothing; thus, this correspondence connection should be removed from the list, leading to take another assumption to handle these cases:

* Nested control-flow structure containing a logging call should be matched with a corresponding structure enclosing another log statement or nothing.

Figure 6 demonstrates the ultimate list of correspondence connection for our examples.

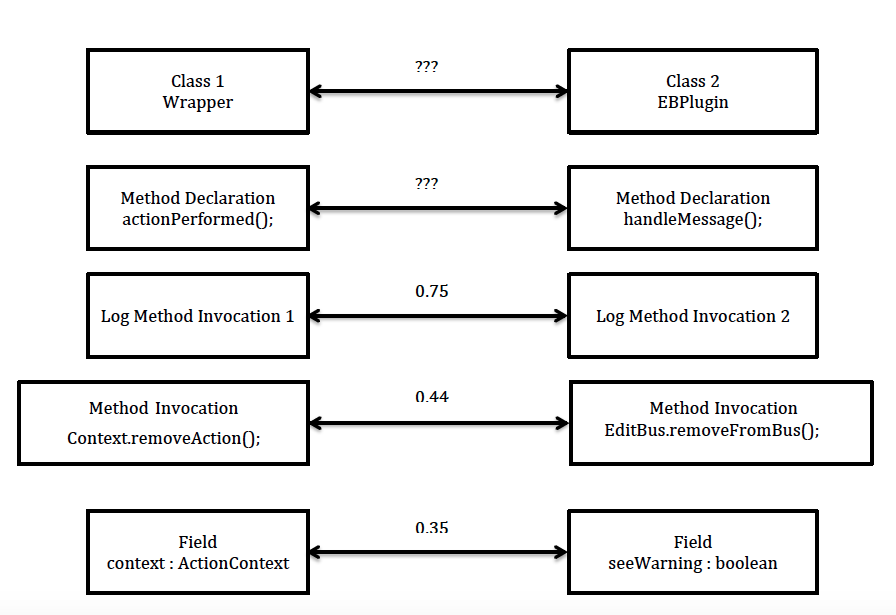
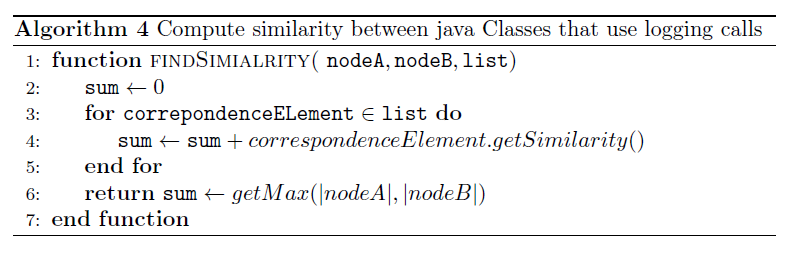


Figure 6: List of correspondence connections

Now we need to computer similarity value between these Java classes to indicate how much similar they are. This value can be an indicator to decide on whether to keep or discard an anti-unifier in later steps. We calculate similarity value using Algorithm 4. It takes an average by summing up similarity value for the list of correspondence connections and dividing it by the number of nodes in the largest AST.

* For the nodes with no correspondence connection in the list we consider zero value for similarity
* For the nodes with one correspondence connection in the list we consider similarity computed by Jigsaw



The similarity between our Java classes would be 0.14 as computed by the following equation:

Class 2

EBPlugin

Class 1

Wrapper

0.14

Method Declaration

handleMessage();

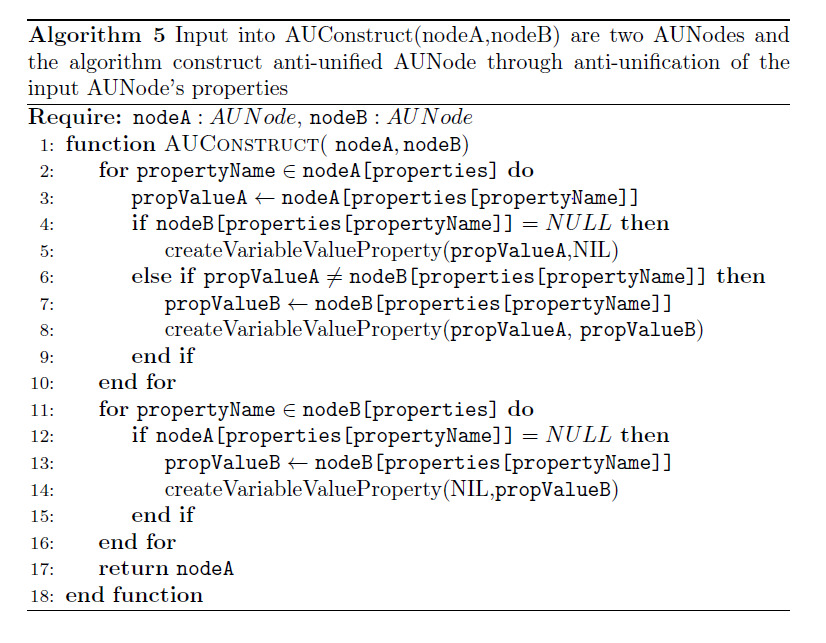
Method Declaration

actionPerformed();

0.31

The third step is to construct an anti-unifier for every correspondence element in the list generated in the previous step; thus, the anti-unifier will eliminate the context around the logging call that does not have a corresponding node in the other seed.

Algorithm 5 demonstrates how an anti-unified AUNode can be constructed from two AUNodes (assume nodeA and nodeB are two corresponding nodes of one correspondence element) through anti-unifying the input node’s properties. Properties of a node are stored in a map data structure that associates each propertyName to its values.



The following will describe this algorithm more precisely:

* Take a pair of AUNodes (e.g. NodeA and NodeB)
* For each property in the nodeA’s list of properties
  + If there is no corresponding property in nodeB
  1. Variable\_propertyValue is created  that can be substituted by “NIL” or nodeA’s propertyValue for that property
  2. Associate Variable\_propertyValue to that specific property of nodeA
  + If there is a corresponding property in nodeB, but with different value
  1. Variable\_propertyValue is created that can be substituted by nodeB’s or nodeA’s propertyValue for the property
  2. Associate Variable\_propertyValue to that specific property of nodeA
* For each property in the nodeB’s list of properties
  + If there is no corresponding property in nodeA
  1. Variable\_propertyValue is created that can be substituted by “NIL” or nodeB’s propertyValue for that property
  2. Add this property to nodeA’s list of properties
  3. Associate Variable\_propertyValue to that specific property of nodeA
* Return nodeA as the anti-unified AUNode

## References

* 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.
* B. Cossette, R. J.Walker, R. Cottrell. Using Structural Generalization to Discover Replacement Functionality for API Evolution, 2014.

1. The pre-determined threshold value for Jigsaw similarity is 0.25, declared by Cottrell. That is, at least one-fourth of the corresponding substructures correspond. [↑](#footnote-ref-1)