Journal of Theoretical and Applied Information Technology
10th January 2014. Vol. 59 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

RECOVERING DESIGN-CODE TRACES: AN ACO BASED APPROACH

[1]IMAD BOUTERAA, [2]NORA BOUNOUR
[1,2] LISCO Laboratory, Badji Mokhtar-Annaba University, P.O. Box 12, 23000 Annaba, Algeria
E-mail: [1]imad.bouteraa@gmail.com, [2]nora_bounour@yahoo.fr

ABSTRACT

Traceability is a key issue to promoting software development quality and productivity. It was recognized as crucial for several software development and maintenance activities. Despite their importance, traceability links are often sacrificed during software evolution due to market pressure. In this work we present an approach to recover them between the design and the implementation. Our approach recover traces basing on properties' similarities, it exploits string edit distance, maximum matching, and Ant Colony Optimization. Evaluations show promising results of this work.

Keywords: *Traceability, Object-Oriented Programming, Software Evolution, Program Understanding, Software Maintenance, UML, Ant Colony Optimization.*

1 INTRODUCTION

Software systems are developed step by step starting from an abstract representation of the system, arriving to a runnable code ready for deployment. Throughout this process, artifacts are created, modified, and refined into more detailed artifacts. This phased nature of the development leaves traces and traceability among artifacts.

According to IEEE Standard Glossary of Software Engineering Terminology [1] traceability is defined as:

i. The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor–successor or master–subordinate relationship to one another.

ii. The degree to which each element in a software development product establishes its reason for existing.

Traceability links are used in many software development tasks, among them we can find:

i. Change impact analysis: this activity aims to identify witch artifacts are affected by a proposed change. When change starts from the code, traceability links help developer to update appropriates high-level artifacts. When change starts from abstract artifacts, traceability links help developers to manage the change and assessing its cost by propagating its impact trough lower levels artifacts.
ii. Requirement Engineering: Many software engineering standards (see [2]) emphasize on the need of requirements traceability to be included into software development process and treat it as a quality factor towards achieving process improvement. It is the key to locate code areas that implement a given requirement and validate that a system meets its requirements.
iii. Program comprehension: When maintaining a legacy system, the first major problem a maintainer is facing may be the comprehension of software system [3]. Trace links can help in both top-down and bottom-up comprehension. In the former, traces give hints on where to look for beacons that either confirm or refute a hypothesis. In the latter one, traces assist programmers in the assignment of a concept to a chunk of code and in the aggregation of chunks into more abstract concepts.
iv. Rational comprehension: traces help developers to understand the rationale behind certain design and implementation aspects of a system.

Lack of traceability can lead to less maintainable software and to defects due to inconsistencies or omissions. It is one of the top factors causing delays in software engineering projects [4]. And it causes the software to deviate from the external quality attributes such as understandability, reusability, and extendibility [5].

However, traceability links are regularly broken and sacrificed during software evolution [6]. This is basically due to them updating cost and the market pressure. Two solutions are envisaged for this problem: reverse engineering, and trace recovering.

Reverse engineering tools can automatically generate a design from the code. Obviously the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

generated design will be consistent with the code and traces can be saved during the design recovery process. But designs produced by users are usually richer than those extracted automatically, since they include context and high level semantic information. Therefore it is always better to use human produced designs, and maintain them consistency with the code.

Trace recovering aims to recuperate traces between artifacts. Manual creation of traceability links between requirements and source code is error-prone, time consuming and complex [7], which emphasize the need of automation of this activity.

In this paper we present a similarity based trace recovering approach. Our approach operates on UML class diagram and the source-code expressed in java. It associates every design class to a set of java classes. We believe that the implementation may split and refine a class from the class diagram into several java classes. Thus, the recovered traces are one-to-many rather than one-to-one.

In the rest of this paper we will refer to classes from a Java program as *implementation classes.* And those form UML class diagram as *design classes*. An implementation class is associated to a design class basing on them properties' similarities. We distinguish two kinds of properties. Elementary properties (e.g. class name, fields' names…etc.) and relationship properties (e.g. inheritance, association… etc.). Similarity computation for the former one is relatively simple; it can be done using string edit distance and a maximum matching algorithm. But with relationship properties we found ourselves in the need of an assumed mapping to compute them. For example, to claim that an implementation class reflects a design class inheritance, we should at least suppose that the implementation class parent is the corresponding of the design class parent. To find the best mapping we use Ant Colony Optimization (ACO) [8] [9], a well-known meta-heuristic that showed its effectiveness in analogous problems.

The rest of this paper is organized as flow; in section 2 we briefly present Ant Colony Optimization since it is a key for this work. Than we present our approach in section 3. And we evaluate it in section 4. After that, we present some related works in section 5. A conclusion and future works are presented in the section 6.

2 ANT COLONY OPTIMIZATION

Ant Colony Optimization (ACO) was introduced by Dorigo, Colorni and Maniezzo [8] in order to resolve the *Traveling Salesman Problem* (*TSP*). The heuristic was inspired from the behavior of real world ants. Experiments have shown that ants can find the shortest path between a food source and their nest.

Basically, every ant starts by seeking randomly for a food. Once it's found, the ant returns to the nest while laying down pheromone trail. If other ants find such path of pheromone, they flow it, and they reinforce it in the returning if they found food. Over time, pheromone starts to evaporate, thus reducing attraction of some trails. The more time an ant takes to travel a path, the more time pheromone evaporates, and the path becomes less and less attractive. A short path gets marched over more frequently, and thus the pheromone density becomes higher on shorter paths than longer ones.

Ant System was the first ACO algorithm. It was presented in [8]. To describe its principle, we take as example the traveling salesman problem. In Ant System every ant tries to construct solution at each iteration. A valuable solution for TSP is a tour that visits each city once. The *m* ants having built a solution in an iteration, update the pheromone values. Pheromone quantity τ_{ij} associated to edge linking city i to j, is updated as a flows:

$$\tau_{ij} \leftarrow (1-\rho)\cdot\tau_{ij} + \sum_{k=1}^{m}\Delta\tau_{ij}^{k} \quad (1)$$

Where ρ is the evaporation rate, m is the number of ants, and $\Delta\tau_{ij}^{k}$ is the quantity of pheromone led on edge (i,j) by ant k:

$$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ used edge } (i,j), \\ 0 & \text{otherwise,} \end{cases} \quad (2)$$

Where Q is a constant, and L_k is the length of the tour constructed by ant k.

When ant k is in city i and has so far constructed the partial solution s^p, the probability of going to city j is given by:

$$p_{ij}^{k} = \begin{cases} \frac{\tau_{ij}^{\alpha}\cdot\eta_{ij}^{\beta}}{\sum_{c_{il}\in N(s^p)}\tau_{ij}^{\alpha}\cdot\eta_{ij}^{\beta}} & \text{if } c_{ij}\in N(s^p), \\ 0 & \text{otherwise,} \end{cases} \quad (3)$$

Where $N(s^p)$ is the set of feasible components; that is, edges (i,l) where l is a city not yet visited by ant k. The parameters α and β control the relative importance of the pheromone versus the heuristic information η_{ij}, which is given by:

$$\eta = \frac{1}{d_{ij}} \quad (4)$$

Where d_{ij} is the distance between cities i and j

3 RECOVERING TRACES

Recovering traceability between design and code can be viewed as a construction of a mapping

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

between the set of classes in class diagrams and the set of classes in source code. The mapping is constructed basing on the similarities between elements of each set.

The process of similarities computation and trace recovery is illustrated by Figure 1. In the rest of this section we will explain the steps of this process.

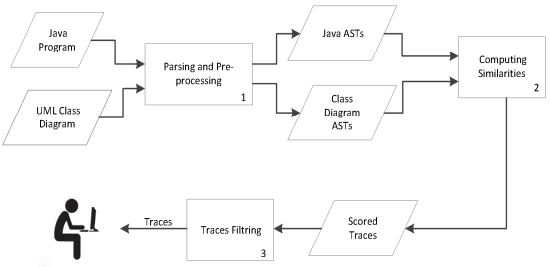


Figure 1: Trace Recovery Process

3.1 Parsing and Pre-processing

In this step we extract and prepare information for similarity assessment. This step takes as input a java program and a UML class diagram; it parses them using respectively a java parser (e.g. eclipse JDT parser) and a UML engine (e.g. Papyrus). This results java ASTs and class diagram ASTs. Then it prepares those ASTs to computation.

Programming languages mostly don't allow the use of whitespaces and punctuation in identifiers. To represent many words in the same identifier, java programmers conventionally [10][11] use two forms: camelCase and separated_by_underscore. From the UML side, there is no naming restriction or convention. This divergence of naming form complicates the task of names comparing. So we must normalize all names by transforming them to a common form. This task is performed as flows:

1. Splitting into separated words the composed identifiers by using an under_score or camelCase separator.
2. Removing non alphabetical characters
3. Converting all uppercase latters into lowercase.

3.2 Computing Similarities

A human can decide if a given class from a code is the correspondent of a design class basing on the shared properties between them. For example if two classes have the same name, at most the same fields names, or at most the same methods names; they are probably classified as mutually corresponding. We use the same principle to compute the similarity between a design class and an implementation class, the global similarity is given by the sum of weighted similarities of their properties divided by the sum of properties weights:

$$\text{similarity}(d,i) = \frac{\sum_{p\in\mathcal{P}} w_p \cdot \text{similarity}_p(d,i)}{\sum_{p\in\mathcal{P}} w_p} \quad (5)$$

Where $\mathcal{P}$ is the set of properties that a class can have, $\text{similarity}_p(d,i)$ the similarity between the design class d and the implementation class i in term of the property p, and w_p is the weight of similarity_p.

We summarize the considered properties in Table 1. We distinguish two categories of property, elementary and relationship. We define an elementary property as a class property that can be described without using other classes. In our case we have three properties in this category; CN, F, and M.

A relationship property for a given class is a property that we describe using other classes. In our case we consider three kinds of properties belonging to this category; *inheritance, dependency,* and *association.*

Table 1: Properties Table

Acronym	Description	Category
CN	Class name	Elementary properties
F	Fields names	
M	Methods names	
I	Inheritances	Relationship properties
A	Associations	
D	Dependencies	

Since the code is the implementation of the design, it has more properties than the design. A property may appear in an implementation class without appearing in the corresponding design class. For such case we consider the property as a detail and its absence in the design don't affect the similarity. We will apply this principal in all similarities that we define.

3.2.1 notations

In the rest of this paper, we consider the following sets:

- $\mathbb{S}$ the set of strings
- $\mathbb{C}$ the set of classes that may exist in UML class diagrams.
- $\mathbb{A}$ the set of fields that a class $c \in \mathbb{C}$ may have
- $\mathbb{M}$ the set of operations that a class $c \in \mathbb{C}$ may have
- $\mathcal{I}nhr \subseteq \mathbb{C} \times \mathbb{C}$ the set of inheritance relationships that may exist in UML class diagrams

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

• $\mathcal{D}epn \subseteq \mathbb{C} \times \mathbb{C}$ the set of dependence relationships that may exist in UML class diagrams
• $\mathcal{A}sso \subseteq \mathbb{C} \times \mathbb{C}$ the set of association relationships that may exist in UML class diagrams
• $\bar{\mathbb{C}}$ the set of classes that may exist in java programs
• $\bar{\mathbb{A}}$ the set of fields that a class $c \in \bar{\mathbb{C}}$ may have
• $\bar{\mathbb{M}}$ the set of methods that a class $c \in \bar{\mathbb{C}}$ may have
We define the following applications:
• $Fields: \mathbb{C} \cup \bar{\mathbb{C}} \to P(\mathbb{A}) \cup P(\bar{\mathbb{A}})$ the application that returns for a given class, all of its fields (P is the power set)
• $Meths: \mathbb{C} \cup \bar{\mathbb{C}} \to P(\mathbb{M}) \cup P(\bar{\mathbb{M}})$ the application that returns for a given class, all of its methods
• $Suppers: \mathbb{C} \cup \bar{\mathbb{C}} \to P(\mathbb{C}) \cup P(\bar{\mathbb{C}})$ the application that returns for a given class, its supper classes

$$Suppers(c) = \begin{cases} \{s \in \mathbb{C}: \exists\, h \in \mathcal{I}nhr, h = (c, s)\} & \text{if } c \in \mathbb{C} \\ \{s \in \bar{\mathbb{C}}: c \text{ inherit directely or indirectly from } s\} & \text{if } c \in \bar{\mathbb{C}} \end{cases}$$

• $Supplrs: \mathbb{C} \to P(\mathbb{C})$ the application that returns for a given class, the set of classes on which it depends.

$$Supplrs(c) = \{s \in \mathbb{C}: \exists\, d \in \mathcal{D}epn, d = (c, s)\}$$

• $Assos: \mathbb{C} \to P(\mathbb{C})$ the application that returns for a given class, the set of its associated classes.

$$Assos(c) = \{s \in \mathbb{C}: \exists\, d \in \mathcal{A}sso, d = (c, s)\}$$

• $Name: \mathbb{C} \cup \bar{\mathbb{C}} \cup \mathbb{A} \cup \bar{\mathbb{A}} \cup \mathbb{M} \cup \bar{\mathbb{M}} \to \mathbb{S}$ the application that returns for a given artifact its name.
• $Type: \bar{\mathbb{A}} \to \bar{\mathbb{C}}$ The application that returns for a given field, the class representing its type.
• $used: \bar{\mathbb{M}} \to P(\bar{\mathbb{C}})$ The application that returns for a given method, the set of classes that it uses as a parameter or variable type.

3.2.2 similarity for atomic properties CN, F, and M

Class names have the nature of string in both design and implementation. So we can use string edit distance (see [12]) to compute the similarity between them. Formally we define η_{cn} the similarity between a design class $d \in \mathbb{C}$ and an implementation class $i \in \bar{\mathbb{C}}$ in term of class name by:

$$\eta_{cn}(d, i) = 1 - \frac{d(name(d), name(i))}{|name(d)| + |name(i)|} \quad (6)$$

Where $d: \mathbb{S}^2 \to \mathbb{N}$ is the edit distance between two strings.

To compute $\eta_f(d, i)$ the similarity between $d \in \mathbb{C}$ and $i \in \bar{\mathbb{C}}$ in term of fields names, we must at first establish a mapping between fields of d and fields of i. The mapping should minimize the amount of editing distances between attribute names. The mapping is calculated by applying a minimum-matching algorithm (see [13]) in a bipartite graph of which nodes are names of fields and edges are weighted by the editing distances between nodes. We note $d(n)$ the weight of the edge connected to node n after the application of the minimum-matching. If n have no corresponding, $d(n)$ is 1. η_f is given by:

$$\eta_f(d, i) = \begin{cases} |Fields(d)| - \sum_{f \in Fields(d)} \frac{d(Name(f))}{|Name(f)|} & \text{if } |Fields(d)| \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

By the same reasoning we put η_m the similarity in term of method names:

$$\eta_m(d, i) = \begin{cases} |Meths(d)| - \sum_{m \in Meths(d)} \frac{d(Name(m))}{|Name(m)|} & \text{if } |Meths(d)| \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

3.2.3 similarity for relationship properties; I, A, and D

We found ourselves unable to compute such similarities without assuming an existing mapping, i.e. we compute relationship similarities basing on a mapping between design and implementation classes. To describe this situation consider the class diagram illustrated by Figure 2, and suppose that we have in the implementation a class defined as "`c4 extends c1`". We can't deduce a heritance correspondence between Class4 from the class diagram and c4 from the code, unless we assume that c1 from the code corresponds to Class1 from the design. So, to decide if there exists a heritance correspondence between design and implementation classes, we must at least suppose a mapping between design supper classes and implementation classes.

In the rest of this paper we consider a mapping map between a set of design classes $\mathcal{D} \subset \mathbb{C}$ and a set of implementation classes $\mathcal{I} \subset \bar{\mathbb{C}}$ as a set of pairs $m \in \mathcal{D} \times \mathcal{I}$. A mapping must verify the following condition:

$$(d, i) \in map \wedge (d', i') \in map \implies d \neq d' \wedge i \neq i'$$

In the rest of this paper, for a given mapping map we note $map(d) \in \mathbb{C}$ the corresponding of $d \in \mathbb{C}$ according to map

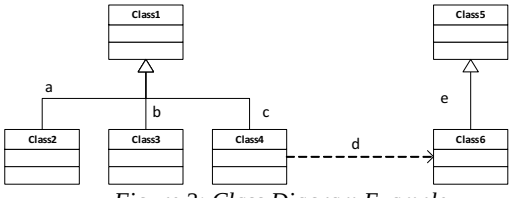


Figure 2: Class Diagram Example

3.2.3.1 *Inheritance similarity*

Inheritance similarity is computed by checking if the design inheritances for a given class are re-

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

flected by its implementing class. All object oriented programming languages have inheritance reserved keywords (e.g. "*extends*" in java). So we can check such similarity by parsing the code.

The inheritance similarity between a design class d and an implementation class i according to the assumed mapping map is given by:

$$L_{inh}(d,i,map) = \begin{cases} \frac{\sum_{e\in\mu_{inh}(d,i,map)} w_{inh}(e)}{|Suppers(d)|} & \text{if } Suppers(d) \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

Where μ_{inh} is the application that returns the set of matched design supper classes. Formally, we define this set by:

$$\mu_{inh}(d,i,map) = \{s: s \in Suppers(d) \wedge map(d) \in Suppers(i)\} \quad (10)$$

And w_{inh} is the originality inheritance weight from a given design class, it is defined by:

$$w_{inh}(d) = \frac{1}{|\{s\in\mathbb{C}: d\in Suppers(s)\}|} \quad (11)$$

3.2.3.2 Associations similarity

We use the same process to compute the association similarity. Generally an association relationship is implemented as a class field. Basing on this principal, we define the association similarity between a design class d and an implementation class i according to the assumed mapping map by:

$$L_{asso}(d,i,map) = \begin{cases} \frac{\sum_{e\in\mu_{asso}(d,i,map)} w_{asso}(e)}{|Assos(d)|} & \text{if } Assos(d) \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

Where μ_{asso} is the application that returns the set of matched design associated classes. Formally, we define this set by:

$$\mu_{asso}(d,i,map) = \{s: s \in Assos(d) \wedge map(s) \in FldsTypes(i)\} \quad (13)$$

$FieldsTypes$ returns the set of fields types for a given implementation class, formally it is defined by:

$$FldsTypes(i) = \cup_{f\in Fields(i)} Type(f) \quad (14)$$

3.2.3.3 Dependencies similarity

Dependency relationship is described as flows: "*A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation*." [14]

According to this definition, if a design class a depends on another class b. This must be reflected in the code by a use of the supplier corresponding class (i.e. the corresponding of b) in the corresponding of a. So, there is two ways to implement a dependency relation; as a type of parameter variable, or as a type of local variable in a method. Association can also be considered as a special kind of dependency, but we ignore this case.

We define the dependence similarity between a design class d and an implementation class i according to the assumed mapping map by:

$$L_{depn}(d,i,map) = \begin{cases} \frac{\sum_{e\in\mu_{depen}(d,i,map)} w_{depn}(e)}{|Supplrs(d)|} & \text{if } Supplrs(d) \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (15)$$

Where μ_{depn} is the application that returns the set of matched suppliers. Formally, we define this set by:

$$\mu_{depn}(d,i,map) = \{s: s \in Supplrs(d) \wedge map(s) \in UsdTypes(i)\} \quad (16)$$

$UsedTypes$ returns the set of used types for a given implementation class, formally it is defined by:

$$FldsTypes(i) = \cup_{m\in meths(i)} used(m) \quad (17)$$

3.2.4 mapping construction

Due to the resemblance between the traveling salesman problem and the construction of a good mapping between design and implementation, we decide to use Ant Colony Optimization [8]. Table 2 summarizes the common points between those two problems.

In order to apply ACO on mapping construction we must define η the distance heuristic, and L the length of a solution. η is computed by combining all atomic similarities, it is given by:

$$\eta(d,i) = \frac{w_{cn}\cdot\eta_{cn}(d,i) + w_f\cdot\eta_f(d,i) + w_m\cdot\eta_m(d,i)}{w_{cn} + \left(w_f\cdot\left(1-0^{|Fields(d)|}\right)\right) + \left(w_m\cdot\left(1-0^{|Methods(d)|}\right)\right)} \quad (18)$$

w_{cn},w_f, and w_m are parameters representing respectively the weights of η_{cn},η_f, and η_m. In the denominator we multiply w_f by $\left(1-0^{|Fields(d)|}\right)$ to eliminate its effect when d has no fields. We use the same mechanism to ignore w_m in the denominator when d has no methods.

Table 2: Common Points Between TSP And Mapping Construction

Travel Selman Problem	Mapping construction
A solution is a cercal that travel all cities	A solution is a mapping that relies every design class to an implementation class
A solution is constructed component by component	A solution is constructed component by component
A component is an edge that relies two cities	A component is an edge that relies a design class to an implementation class
The quality of a component is computed basing on the distances between cities	The quality of a component is computed basing on atomic similarities

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

	between the design and the implementation class
The qualities of components can guide the construction of the best solution	The qualities of components can guide the construction of the solution
The quality of a solution is computed using qualities of all components (i.e. distances)	The quality of a solution is computed using qualities of all components (i.e. similarities)

Once we construct a mapping map that maps every design class to an implementation class, we can compute relationship similarities and introduce them in similarity formula. We define the global similarity between a design class d and an implementation class i, or the score of the trace from d to i according to the assumed mapping map by:

$$L(d,i,map) = \frac{w_\eta \cdot \eta(d,i) + w_{inh} \cdot L_{inh}(d,i,map) + w_{asso} \cdot L_{asso}(d,i,map) + w_{depn} \cdot L_{depn}(d,i,map)}{w_\eta + \left(w_{inh} \cdot (1 - 0^{|Suppers(d)|})\right) + \left(w_{asso} \cdot (1 - 0^{|Assos(d)|})\right) + \left(w_{depn} \cdot (1 - 0^{|Supplrs(d)|})\right)} \quad (19)$$

Where w_η,w_{inh}, w_{asso}, and w_{depn} are parameters.

Now we can assess the global quality of the mapping. The quality (length in TSP) of a mapping map that maps a set of design classes $\mathcal{D}$ to a set of implementation classes is given by:

$$L(\mathcal{D}, map) = \frac{\sum_{d \in \mathcal{D}} L(d, map(d), map)}{|\mathcal{D}|} \quad (20)$$

After defining all the needed similarities, we can apply ACO meta-heuristic on the problem of mapping construction. In our approach, ants seek for the best mapping by constructing mappings and assessing them qualities. The construction of a mapping is a progressive task; it's done component by component. A component is pair $(d,i) \in \mathbb{C} \times \bar{\mathbb{C}}$ indicating that the design class d is implemented in the code as i. To construct a mapping between $\mathcal{D} \subset \mathbb{C}$ and $\mathcal{I} \subset \bar{\mathbb{C}}$, ants flow the algorithm in Figure 3.

```
1  while (stop condition not reached)
2   map = enmpty set
3   for each d in 𝒟
4    choose i a valid corresponding of d from ℐ
5    save (d,i) into map
6   rof
7   update pheromone basing on map
8   if (distance(map) > distance(bestMap))
9    bestMap = map
10  fi
11 end while
```

Figure 3: Mapping construction algorithm

Where $bestMap$ is an initially empty shared variable. The *stop condition* – in the first line – can be defined as a number of iterations, the stability of the results, or a reaching of predefined threshold by the $bestMap$ distance.

A valid corresponding – in the fourth line – for a design class according to a partial mapping map is an implementation class that never be used in all ordered pairs belonging to map. The probability of choosing $i \in \mathcal{I}$ as the corresponding of $d \in \mathcal{D}$ according to the partial mapping $map \in \mathrm{P}(\mathcal{D} \times \mathcal{I})$ is given by formula 21

$$P_{di}^{map} = \begin{cases} \frac{\tau_{di}^\alpha \cdot \eta(d,i)^\beta}{\sum_{k \in \mathrm{N}(\mathcal{I}, map)} \tau_{dk}^\alpha \cdot \eta(d,i)^\beta} & \text{if } i \in \mathrm{N}(\mathcal{I}, map), \\ 0 & \text{otherwise,} \end{cases} \quad (21)$$

$\mathrm{N}(\mathcal{I}, map)$ is the set of valid implementing classes according to the partial mapping map, it is defined by:

$$\mathrm{N}(\mathcal{I}, map) = \{i \in \mathcal{I} : \forall (d,k) \in map \Rightarrow k \neq i\} \quad (22)$$

When the program reaches the seventh line, a valid solution (i.e. a mapping) should be constructed. The pheromone updating policy is based on the quality of the solution. We define the new pheromone value between $d \in \mathcal{D}$ and $i \in \mathcal{I}$ according to the solution $map \in \mathrm{P}(\mathcal{D} \times \mathcal{I})$ by:

$$\tau_{di}^{map} = \begin{cases} (1-\rho) \cdot \tau_{di} + Q \times L(\mathcal{C}, map) & \text{if } (d,i) \in map \\ (1-\rho) \cdot \tau_{di} & \text{otherwise,} \end{cases} \quad (23)$$

Where ρ is the evaporation rate, Q is a constant, τ_{di} is the amount of pheromone that represents the attraction of choosing i as the corresponding of d

3.2.5 trace scoring

From a theoretical perspective, traces exist between every design class and every implementation class. But some are relevant and others aren't. We score each trace by its relevance degree. The relevance degree of a trace is given by formula (19) using the best mapping constructed previously by ACO.

3.3 Trace Filtering

In this step we use a threshold to accept or reject traces. The decision is taken by comparing traces scores to a threshold. The threshold can be selected by the user or computed automatically using Zhao method [15]. If we apply Zhao method on our work, we can automatically determine the threshold for each design class d by sorting its traces in descending order according to their scores. Then we compute the differences between each two successive scores, and we identify the two traces

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

having the greatest distance. The score ε of the upper one is used as a threshold. All traces that trace d having score greater than or equal to ε will be considered as accepted traces.

It's worth noting that our approach may accept for the same design class many corresponding implementation classes. Two interpretations are possible for such case, the former is that the design class is refined into many implementation classes; the latter is that the scores are to contiguous for automatic filtering and they require human intervention.

4 CASE STUDY

4.1 Subject

We use as subject of our experiment an ATM system. A well-known example in object oriented development. We got the design and the Java implementation from [16] it's a pedagogic example that aims to teach the object oriented paradigm.

Due to its pedagogic purpose, the design of our subject fits perfectly with its implementation. So if we take the original versions of the code and the design, we will be able to perfectly recover all traces by only using class names. To make our subject more realistic, we modify some names in the source code. The modification is basically a translation from English to French. This may accrue in the case where programmers are originally francophone. Table 3 represents statistics about the ATM system design and implementation.

Table 3: ATM statistics

	Artifact type	Occurrence count
Implementation	Line of code	2418
	Constructors	39
	Fields	174
	Methods	151
	Packages	6
	Classes	65
Design	Classes	22
	Fields	108
	Operations	97
	Associations	17
	Inheritances	4
	Dependencies	17

4.2 Metrics

To measure the quality of results we use the two most frequent and basic measures for information retrieval effectiveness, *Recall* and *Precision* [17]. *Precession* measures the fraction of relevance of the returned results to the need information. *Recall* measures the fraction of the relevant results that are retrieved.

Formally precision and recall are defined as:

$$\text{precision}(d) = \frac{|relevant(d) \cap matched(d)|}{|matched(d)|}$$

$$\text{recall}(d) = \frac{|relevant(d) \cap matched(d)|}{|relevant(d)|}$$

4.3 Results

To illustrate the gain that relationships bring, we compare the results of our approach with those given by formula (18) that combines all atomic similarities. Table 4, Table 5, and

Table 6 represent respectively the returned results for the predefined thresholds 0.70 and 0.50 and for the automatic threshold. In every one of those tables, *Classes* column represents the design classes of our subject, It contains all classes that exist in class diagrams of ATM system; η column represents the results assessed by the similarity that combines all atomic similarities; And L column represents the results returned by our approach.

Table 4: Results For The Threshold 0.70

Classes	Recall		Precision	
	η	L	η	L
Money	1	1	1	1
Balances	0	0	−	−
ATM	0	0	−	−
Log	0	0	−	−
Cash Dispenser	0	0	−	−
Message	1	0	1	−
Envelope Acceptor	1	1	1	1
Card Reader	1	0	1	−
Network ToBank	1	0	1	−
Status	1	1	1	1
Customer Console	1	1	1	1
Receipt	0	0	−	−
Transaction	1	0	1	−
Card	1	1	1	1
Receipt Printer	1	1	0.5	1
Operator Panel	1	1	1	1
Session	1	0	1	−
Withdrawal	1	0	1	−
Inquiry	0	0	−	−
Transfer	1	0	1	−
Deposit	1	0	1	−
Account Information	1	1	1	1
Total	0.72	0.36	0.94	1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 5: Results For The Threshold 0.50

Classes	Recall		Precision	
	η	L	η	L
Money	1	1	1	1
Balances	0	0	–	–
ATM	1	1	1	1
Log	1	1	0.5	1
Cash Dispenser	0	0	–	–
Message	1	1	1	1
Envelope Acceptor	1	1	0.33	1
Card Reader	1	1	1	1
Network ToBank	1	0	1	–
Status	1	1	0.25	0.25
Customer Console	1	1	1	1
Receipt	1	0	1	–
Transaction	1	0	1	–
Card	1	1	1	1
Receipt Printer	1	1	0.33	1
Operator Panel	1	1	1	1
Session	1	0	0.5	–
Withdrawal	1	0	1	–
Inquiry	0	0	–	–
Transfer	1	0	1	–
Deposit	1	0	1	–
Account Information	1	1	0.09	0.09
Total	0.86	0.54	0.5	0,48

Table 6: Results For Zhao Threshold

Classes	Recall		Precision	
	η	L	η	L
Money	1	1	1	1
Balances	1	1	1	1
ATM	1	1	1	1
Log	1	1	0.5	1
Cash Dispenser	1	1	1	1
Message	1	1	1	1
Envelope Acceptor	1	1	1	1
Card Reader	1	1	1	1
Network ToBank	1	1	1	1
Status	1	1	1	1
Customer Console	1	1	1	1
Receipt	1	1	1	1
Transaction	1	1	1	1
Card	1	1	1	1
Receipt Printer	1	1	1	1
Operator Panel	1	1	1	1
Session	1	1	1	1
Withdrawal	1	1	1	1
Inquiry	1	1	0.16	0.25
Transfer	1	1	1	1
Deposit	1	1	1	1
Account Information	1	1	1	1
Total	1	1	0.78	0.88

4.4 Discussion

From Table 4 we could see that when we use the predefined threshold 0.70 method (i.e. the one based on atomic similarities) retrieves more traces than our approach. Our approach seems very weak since it retrieves only 36% of traces. Comparing to witch retrieves 72% of traces we can claim that relationships similarities decrease the quality of results when we use 0.70 as a threshold. This can be interpreted in two ways, i) as a disadvantage of introducing relationship similarities; ii) or as a decreasing of similarities when we consider relationships. According to the second interpretation, we suppose that we can get better results using smaller threshold and that's what leads as to results of

Table 5. We will discuss results of this last one after talking about the *precision*. As Table 4 reports, the two considered methods give a high level of precision. Results of η method are 94% precise and those of our approach are 100% precise. The perfect precision of our approach when we use the threshold 0.70 is not very significant because of its low recall.

In

Table 5 we can see that when we use of threshold 0.50 both of methods return a better recall. η method retrieves 86% of traces and our approach retrieves 54% of traces. Despite the improvement of our approach's recall (i.e. from 36% to 54%), it remains useless. The least we can say, 54% is insignificant comparing to 86% retrieved by η method. From the precision point of view, η method returns also better results than ours. η method is precise at 50% and our approach is at 48%. But for both methods, returning results precise at almost 50% is not useful as an automated trace recovering technique.

Choosing the right threshold is a challenge. We can see that for both methods, if we increase the threshold, we gain more precision but we lose some recall. And if we decrease it, we gain more recall and we lose precision. The reason is that the similarity between a design class and its relevant corresponding implementation class change from a class to another. This is confirmed by Figure 4 which represents similarities between design classes and there relevant correspondents. From Figure 4 we can see that there is no good common threshold. Even if we choose the average, we will get bad results because of the important similarity variance.

Journal of Theoretical and Applied Information Technology
10th January 2014. Vol. 59 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

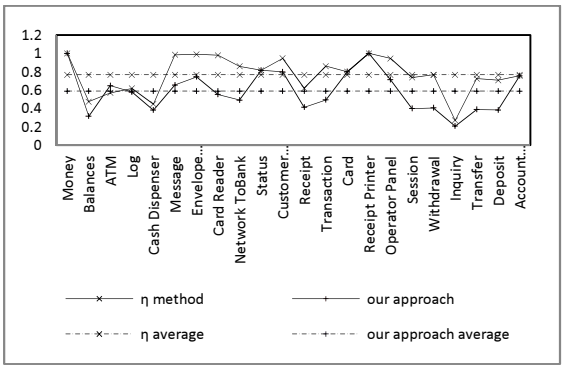


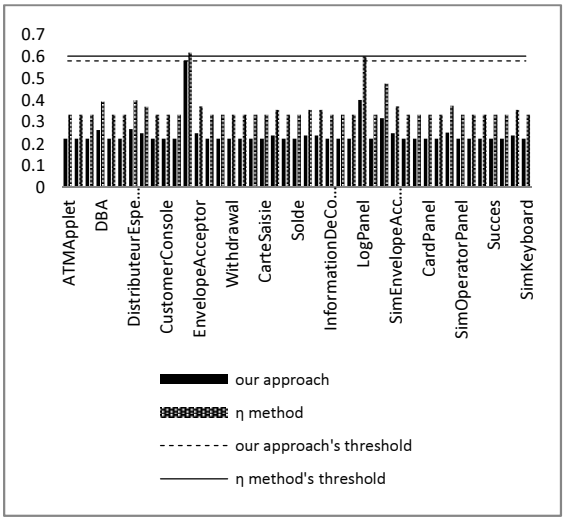
Figure 4: Similarities Between Design Classes And There Relevant Correspondents

Due to the bad quality that we got when we used common thresholds, we must specify a threshold for each design class. So we use Zhao method to assess the right threshold.

Table 6 represents the gotten results using an automatic threshold assessment. From that table we can see that both of methods got a perfect recall (i.e. 100%). In term of precision, results of our approach are precise at 88%, and those of η method are at 78%. We could claim that our approach is precise than η method, and relationships help us to eliminate false positive traces.

From

Table 6 we could see that our approach reduce false positives traces in the case of *Log* and the case of *Inquiry*. To details the effectiveness of our approach in the case of *Inquiry* and *Log*. We drive the histograms of Figure 5 and Figure 6. They represent the traces' scores and Zhao thresholds respectively for *Inquiry* and for *Log*.



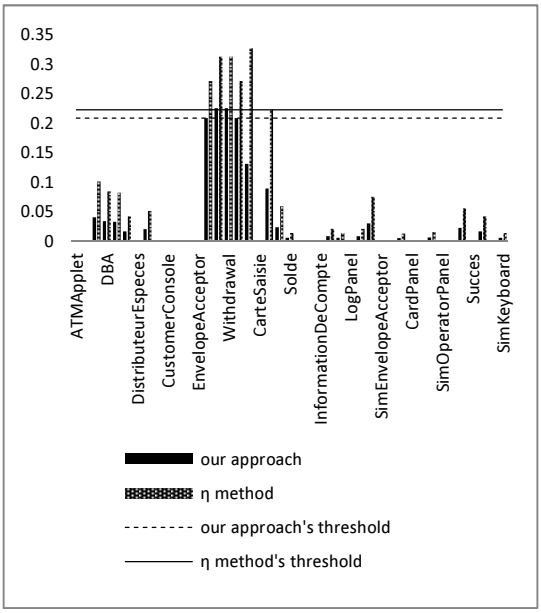


Figure 5: Traces' Scores For Inquiry Class

Figure 6: Traces' Scores For Log Class

5 RELATED WORKS

A lot of papers treat the problem of trace recovering. The iterative nature of software development breeds lot of kinds of traces. In the literature several works were proposed to recover those traces by focusing on specific kinds. We identify two main categories of trace recovery approaches. Those basing on Information Retrieval (IR) technics, and those how are basing on properties' similarities.

5.1 IR based Approaches

In [18] authors recover traces between free text documents and source code. They exploit the idea of language model, i.e. a stochastic model that assigns a probability value to every string of words taken from a prescribed vocabulary. Language models are estimated from the documents, one for each document or identifiable section. Then they apply Bayesian classification to score the sequence of mnemonics extracted from a selected area of code against the language models. A high score indicates a high probability that a particular sequence of mnemonics drives from the document or document's section that generated the language

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

model. In the context of information retrieval, this approach is classified as a probabilistic approach.

Then Antoniol et al. [19] exploit VSM (Vector Space Model) to recover traceability links between source code and free text. The request for a given class is composed of identifiers belonging to that class. And similarity between the request and the document is computed by cosine.

In [20] authors compare the probabilistic model presented in [18] and the vector space model presented in [19]. The results show that the former model is mostly better than the latter in term of recall and precision.

De Lucia [21], [22] propose an approach based on LSI (Latent Semantic Indexing) to recover traces between different kinds of artifacts (e.g. requirement, design, test case, and code). The approach is implemented in a system named ADAMS.

Wang et al. [23] enhance the LSI method to recover more relevant traces between high level documents and source code. The paper presents four enhancements: source code clustering, identifier classifying, similarity thesaurus, and hierarchical structure enhancement.

De Lucia et al. [24] propose an approach to help developers to maintain source code identifiers and comments consistent with high-level artifacts. The approach use LSI to assess the similarity between low-level and high-level artifacts. It was implemented in an eclipse plug-in named COCONUT. In addition to similarity computation, the tool suggests identifiers obtained by extracting n-grams form the high-level documents.

Another IR based approach was proposed in [25]. In that work, authors use semantic relatedness to recover traceability links between free-text documents and source-code. Authors use a Wikipedia-based semantic relatedness measure, namely Explicit Semantic Analysis (ESA) [26] as SR retrieval technique.

Textual and structural information are combined in [27] to recover traceability links. Authors use LSI to recover *document-to-source* traces, and JRipples [28] to recover *source-to-source* traces. Then they infer *document-to-document* traces basing on the previous results. Although such a combination increase the recovered links, it has an important disadvantage. False positive links recovered by IR will significantly pollute the results by inferring false positive links. To overcome this drawback, an approach was recently proposed in [29], basically the improvement consists of validating IR recovered links before inferring new ones.

IR based approaches don't exploit the structural aspect of the code and the design. They are rather adapted to free text. It's worth noting that those approaches suffer from the same problems of IR, i.e. polysemy (different meanings for the same word) and synonymy (same meaning for different words). The LSI technique was proposed to overcome these failings, but it is unable in front of situations where the artifacts are written using different lexicons. Which is very probable in software development since several developers gets involved in this task.

5.2 Properties' Similarities based Approaches

Antoniol et al. [30], [31] recover traces between design and code using the edit distance computation and the maximum match algorithm. Global similarity between two artifacts is computed basing on the resemblance of their names, their fields names, and their methods names and signatures.

Then Antoniol et al. [32] extend the work in [30] by introducing relationship and dictionnary similarities. This work differs from ours since it uses relationships to check the quality of tractability rather than using them in the construction of traces.

In [33] authors asses the similarity between a design artifact and an implementation artifact basing on three kinds of similarities: classifiers names, metric profiles, and packages.

Classes in design and in implementation share lot of properties. Which properties are more relevant to trace recovering is the subject of the research presented by Antoniol et al. [34]. In that work, different combinations where analyzed. All the gotten results suggest that the best performing combinations are those with an explicit representation of the class name, while very poor performances are associated to methods based on the relations of a class with the other classes.

It is worth noting that this work use relationships in a different manner than us. They transform it to a class attributes of type string. If, for example, class *A* generalizes class *B*, the attribute *"generalizes->B"* is attached to class *A*, and the attribute *"extends->A"* is attached to class *B*. This is not the best way to exploit relationships in trace recovery, since all relationships having the same type will have the same common prefix (e.g "generalize->"), which disturbs the similarity. To illustrate this case, take the example of "*generalize->pen"* and "*generalize->cat*", the similarity between those two strings basing on edit distance is at 90%, which is far higher than the reality. Another disadvantage of this method is that the similarity depends only on the names of the classes.

6 CONCLUSION

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Traceability links helps in terms of quality and productivity. They are generally involved in several activities of software development, and especially in maintenance. But market and delays pressures force developers to sacrifice them. The purpose of this work is to recover traces between design and implementation.

This work describes a similarity based approach to recover traceability links between UML design classes and java classes. The similarity is computed basing on properties resemblance. Two categories of properties are considered: elementary properties and relationship properties. Resemblance for the former is assessed using string edit distance and a maximum matching algorithm. To compute the similarity in term of a relationship property, we must assume a mapping between the design and the code. The mapping is constructed using an ACO algorithm.

The originality of this work consists in the exploitation of relationships. When computing similarity in terms of a relationship, we consider the nature of the relationship, and all the property of the class that is at the other end of the relationship. According to the evaluation result, we can see that this approach seems promising. Experimentation shows that, when we use Zhao threshold, relationships improve to 10% the precision while keeping the same recall. Evaluation shows also that our approach gives bad results when using a predefined threshold. Thus we strongly recommend the use of Zhao threshold to filter traces.

The case study subject is not large enough to evaluate this work, but the lack of suitable subjects on the internet has forced us to take such a decision. In future work we will extend our approach to cover other programing languages and diagrams. An evaluation on larger subjects is also planned and will be the subject of future works. It is worth noting that this work is part of a large change impact analysis project.

REFERENCES:

[1] IEEE Std 610.12, IEEE Standard Glossary of Software Engineering Terminology, (1990)

[2] ISO/IEC 12207, Software & Systems Engineering Standards Committee of the IEEE Computer Society, (2008)

[3] Zhao W., Zhang L., Liu Y., Luo J., & Sun J.: Understanding how the requirements are implemented in source code. In Software Engineering Conference, Tenth Asia-Pacific, (December 2003), pp. 68-77.

[4] Borg M.: IR-based Traceability Recovery as a Plugin–An Industrial Case Study. In Proceedings of the Fourth BCS-IRSG conference on Future Directions in Information Access. (2011), pp. 14-17

[5] Shatnawi R., Alzu'bi A.: A Verification of the Correspondence between Design and Implementation Quality Attributes Using a Hierarchal Quality Model. IAENG International Journal of Computer Science, Vol. 38, No. 3, (2011), pp. 225-233.

[6] Hammad M., Collard M. L., & Maletic J. I.: Automatically identifying changes that impact code-to-design traceability during evolution. Software Quality Journal, Vol. 19, No. 1, (2011), pp. 35-64.

[7] Delater A., Narayan N., & Paech B.: Tracing Requirements and Source Code during Software Development. In ICSEA 2012, The Seventh International Conference on Software Engineering Advances, (2012), pp. 274-282.

[8] Dorigo M., Maniezzo V., Colorni A., & Maniezzo V.:Positive feedback as a search strategy. (Tech. Rep. 91-016). Milan, Italy: Politecnico di Milano, Dipartimento di Elettronica, (1991).

[9] Colorni A., Dorigo M., & Maniezzo V.: Distributed optimization by ant colonies. In Proceedings of the first European conference on artificial life, (1991), pp. 134-142.

[10]: IBM, “Java naming conventions,” available at http://www.ibm.com/developerworks/library/ws-tip-namingconv.html (last visited on 24/08/13)

[11]: Oracle, “Code Conventions for the Java Programming Language,” available at http://www.oracle.com/technetwork/java/codeconv-138413.html (last visited on 24/08/13)

[12] Gusfield D.: Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press. (1997)

[13] Cormen T. H., Leiserson C. E., Rivest R. L., & Stein C.: Introduction to algorithms. Third edition. The MIT press. (2001)

[14] OMG Unified Modeling LanguageTM (OMG UML),Superstructure, Version 2.4.1, Date: August 2011

[15] Zhao W., Zhang L., Liu Y., Sun J., & Yang F.: SNIAFL: Towards a static noninteractive approach to feature location. ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 15, No. 2, (2006), pp. 195-226.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[16]: http://www.math-cs.gordon.edu/courses/cs211/ATMExample/ (last visited on 24/08/13)

[17] Manning C. D., Raghavan P., & Schütze H.: Introduction to information retrieval. Cambridge: Cambridge University Press, (2008)

[18] Antoniol G., Canfora G., De Lucia A., & Merlo E.: Recovering code to documentation links in OO systems. Reverse Engineering. Proceedings. Sixth Working Conference on., (1999), pp. 136-144.

[19] Antoniol G, Canfora G., Casazza G., et al.: Information retrieval models for recovering traceability links between code and documentation. In : Software Maintenance, 2000. Proceedings. International Conference on. IEEE, (2000). pp. 40-49.

[20] Antoniol G., Canfora G., Casazza G., & De Lucia A.: Recovering traceability links between code and documentation. Software Engineering, IEEE Transactions on, Vol. 28, No. 10 (2002), pp. 970-983.

[21] De Lucia A., Fasano F., Oliveto R., & Tortora G.: Enhancing an artefact management system with traceability recovery features. In Software Maintenance Proceedings. 20th IEEE International Conference on, (2004), pp. 306-315.

[22] De Lucia A., Fasano F., Oliveto R., & Tortora G.: Adams re-trace: A traceability recovery tool. In Software Maintenance and Reengineering, CSMR 2005. Ninth European Conference on, (2005) pp. 32-41.

[23] Wang X., Lai G., & Liu C.: Recovering relationships between documentation and source code based on the characteristics of software engineering. Electronic Notes in Theoretical Computer Science, Vol. 243, (2009), pp. 121-137.

[24] De Lucia A., Di Penta M., & Oliveto R.: Improving source code lexicon via traceability and information retrieval. Software Engineering, IEEE Transactions on, Vol. 37, No. 2, (2011), pp. 205-227.

[25] Mahmoud A., Niu N., & Xu S.: A semantic relatedness approach for traceability link recovery. In Program Comprehension (ICPC), 2012 IEEE 20th International Conference on, (2012, June), pp. 183-192.

[26] Gabrilovich E., & Markovitch S.: Computing Semantic Relatedness Using Wikipedia-based Explicit Semantic Analysis. In IJCAI, 7, (2007, January), pp. 1606-1611.

[27] McMillan C., Poshyvanyk D., & Revelle M.: Combining textual and structural analysis of software artifacts for traceability link recovery. In Traceability in Emerging Forms of Software Engineering, 2009. TEFSE'09. ICSE Workshop on, (2009, May), pp. 41-48.

[28] Buckner J., Buchta J., Petrenko M., & Rajlich V.:. JRipples: A tool for program comprehension during incremental change. In Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on, (2005, May), pp. 149-152.

[29] Panichella A., McMillan C., Moritz E., Palmieri D., Oliveto R., Poshyvanyk D., & De Lucia A.: When and how using structural information to improve IR-based traceability recovery. In Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on , (2013, March), pp. 199-208.

[30] Antoniol G., Potrich A., Tonella P., & Fiutem R.: Evolving object oriented design to improve code traceability. In. Proceedings. Seventh International Workshop on Program Comprehension (1999), pp. 151-160.

[31] Antoniol G., Canfora G., Casazza G., & De Lucia A.: Maintaining traceability links during object-oriented software evolution. Software: Practice and Experience, Vol. 31, No. 4, (2001), pp. 331-355.

[32] Antoniol G., Caprile B., Potrich A., & Tonella P.: Design-code traceability for object-oriented systems. Annals of Software Engineering, Vol. 9. No. 1-2, (2000): 35-58.

[33] Van Opzeeland D. J., Lange C. F., & Chaudron M. R.: Quantitative techniques for the assessment of correspondence between UML designs and implementations. In 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, (2005)

[34] Antoniol G., Caprile B., Potrich A., & Tonella P.: Design-code traceability recovery: selecting the basic linkage properties. Science of Computer Programming, Vol. 40, No. 2, (2001), pp. 213-234.