# Traceability Recovery in RAD Software Systems

M. Di Penta, S. Gradara, G. Antoniol
dipenta@unisannio.it, gradara@unisannio.it, antoniol@ieee.org

RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo Bosco Lucarelli, Piazza Roma 82100 Benevento, Italy

## Abstract

*This paper proposes an approach and a process to recover traceability links between source code and free text documents in software system developed with extensive use of COTS, middleware, and automatically generated code.*

*The approach relies on a process to filter information gathered from low level artifacts. Information filtering was performed according to a taxonomy of factors affecting traceability links recovery methods. Those factors were directly stem from software rapid development techniques.*

*The approach was applied to recover traceability links from a industrial software, developed with RAD techniques and tools, and making use of COTS (e.g., database access components), automatically generated code (e.g., via GUI builder and report generators), and middleware (i.e., CORBA). Results are presented, along with lessons learned.*

**Keywords: traceability, RAD, COTS, program understanding**

## 1. Introduction

Software mass market production forces the adoption of techniques and approaches aiming at increasing the software productivity. The huge pressure to increase productivity promotes approaches derived from manufacturing, such as components and standardizations; the underlying assumption is that software industry may improve productivity much alike to manufacturing by applying the same or similar approaches. Albeit the adoption of reuse, Commercial Off The Shelf (COTS) components, code generators, middleware, frameworks, Rapid Application Development (RAD) tools and techniques, software industry is still far from the promulgated productivity increase.

Meanwhile, the pressure to reduce time to market, increasing productivity, results in process tailoring, often sacrificing *unproductive* activities. Establishing and maintaining traceability and consistency between software artifacts is one of those costly and tedious activity frequently neglected. However, traceability links between the free text documentation associated with the development and maintenance cycle of a software system and its source code is helpful in a number of tasks, such as program comprehension and software maintenance.

Existing cognition models share the idea that program comprehension occurs in a bottom-up manner [21, 22], a top-down manner [11, 24], or some combination of the two [19, 28, 29, 30]. They also agree that programmers use different types of knowledge during program comprehension, ranging from domain specific knowledge to general programming knowledge [11, 23, 27]. Traceability creates links between areas of code and related sections of free text documents, such as an application domain handbook, a specification document, a set of design documents, or manual pages, aiding both top-down and bottom-up comprehension. Traceability links may also help in design recovery, as defined in [12], a preliminary activity required for the purpose of maintaining any legacy systems. Design recovery requires different sources of information, such as source code, design documentation, personal experience and general knowledge about problem and application domains [8, 20]. Central to design recovery is the recovery of *higher-level abstractions beyond those obtained by examining the system itself* [12], and traceability links may ease the task of identifying such abstractions.

In a previous work [6], an approach to recover traceability links based on the conjecture of the *sane programmer* was presented. The underlying assumption is that *sane programmers* tend to process application-domain knowledge in a consistent way: program item names of different code regions related to a given high-level concept are likely to be the same or at least very similar. Under the above assumption, the knowledge of existing traceability links can be exploited; once programmer behavior can be modeled, few links suffice to allow the method bootstrapping itself.

However, in applying the method proposed in [6] to industrial software we have identified several affecting factors, related to techniques, tools and approaches adopted to increase productivity, that may have threatened traceability link recovery.

In this paper, these factors are identified, discussed and classified. Moreover, compared to [6], the paper proposes an approach and a process, based on the *sane programmer* conjecture and the equations presented in [6], to recover traceability links in software systems developed with an extensive use of techniques aiming at reducing time to market. As in [6] the approach relies on Bayesian classification; however, it requires a process to filter information, accounting for factors affecting the traceability link recovery. For example, a GUI, automatically generated by a GUI builder tool, contains identifiers and comments not necessarily related to the high-level documents. One of the advocated advantages of methodologies centered on RAD and iterative prototyping is that each iteration delivers a functional version of the final system. The drawback is that automatically generated code is poorly traced to high-level documents, thus correspondence between high-level artifacts and source code prototypes may be very difficult to obtain and to maintain. The proposed process filters information gathered from low-level artifacts as well as from high-level documentation, attempting to discard those items with low or null traceability, items that decrease the traceability recovery process accuracy.

The approach was applied to recover traceability links in the first release of an industrial software developed with COTS components (e.g., database access components), automatically generated code (e.g., by GUI builders or report generators), and middleware (CORBA). A relevant fraction of the system was developed with a RAD tool following an iterative prototyping development process. Results are presented along with lessons learned. The development team was asked to produce a traceability mapping between requirements and C++ code. Traceability link recovery accuracy was evaluated with respect to that mapping.

The remainder of the paper is organized as follows. Section 2 summarizes background notions and equations presented in [6]. Section 3 introduces the traceability links affecting factors, while Section 4 presents the process to filter the information to recover traceability links. Tools and case study are described in Sections 5 and 6 respectively. Experimental results and lessons learned are discussed in Sections 7 and 8. Related works are discussed in Section 9; finally, Section 10 gives concluding remarks and outlines directions for future work.

## 2. Background Notions

Unlike most reverse engineering problems, recovering traceability links between free text documentation and source code components cannot be simply based on techniques derived from the compilers' field, because of the difficulty of applying syntactic analysis to natural language sentences.

Our approach exploits techniques widely used in other areas, such as information retrieval, information theory, and speech recognition. Indeed, the problem of identifying the documents related to a given source code component can be seen as a typical information retrieval or pattern recognition task.

### 2.1. Mapping Equation

In the software development/maintenance life cycle programmers may be thought of as stochastic channels transforming a high-level text document into *observations* i.e., chunks of code. More precisely, development/maintenance activity is modeled by a stochastic channel, where the sequence of words (a high-level text document) $W_k$, generated by a source $S$ with the *a-priori* probability $Pr(W_k)$, is transmitted through a channel and transformed into the sequence of observations O with probability $Pr(O \mid W_k)$.

The problem is decoding the observation Y, e.g., program item names, into the original high-level document $W_k$: that is, finding $\hat{W}_k$ that maximizes the *a-posteriori* probability $Pr(W_k \mid O)$. Applying the Bayes rule, we obtain the following identity:

$$Pr(W_k \mid O) = \frac{Pr(O \mid W_k) Pr(W_k)}{Pr(O)} \qquad (1)$$

The probability $Pr(W_k)$ is the high-level document *a-priori* probability and it can be safely assumed that documents are equally likely: in fact there is no reason to believe that a high-level text document is more likely than any other (i.e., there is no *a-priori* information on the document distribution); furthermore, in the above equation, $Pr(O)$ is a constant with respect to $k$. By eliminating the constant factor, $Pr(O)$ and $Pr(W_k)$ decoding is equivalent to find:

$$\hat{W}_k = arg \max_{W_k} Pr(O \mid W_k) \qquad (2)$$

Let us assume that a high-level text document is represented as $W_k = \bigcup_{i=1}^{n} \{w_{k,i}\}$; the document is composed of $n$ words: $w_{k,1}, \ldots, w_{k,n}$ assumed to be independent events. The high-level document $W_k$ is transformed by programmers into an observation $O = \bigcap_{j=1}^{m} \{o_j\}$ corresponding to a low-level artifact (e.g., the identifier collection extracted from a chunk of source code).

Under quite general assumptions [6] the following expression of $Pr(O|W_k)$ is obtained:

$$Pr(O|W_k) = \prod_{j=1}^{m} \sum_{i=1}^{n} Pr(o_j|w_{k,i}) Pr(w_{k,i}) \qquad (3)$$

Equations (2) and (3) are central to the approach, they provide a means to effectively recover traceability links based on the *a-priori* knowledge of a subset of traceability links.

$Pr(o_j|w_{k,i})$ and $Pr(w_{k,i})$ need to be estimated on a *labeled* training corpus, a subset of the high-level documentation and code fragments known to belong to treaceability relations. Given the labeled training corpus, probabilities are approximated with frequencies:

$$Pr(o_j|w_{k,i}) \simeq \frac{c(o_j w_{k,i})}{c(w_{k,i})} \; i = 1 \ldots n \; j = 1 \ldots m \qquad (4)$$

$$Pr(w_{k,i}) \simeq \frac{c(w_{k,i})}{|W_k|} \qquad (5)$$

where $c(h)$ is the number of times that the $h$ word appears in the texts, in the same way $c(hl)$ is the number of times that the couple $h, l$ appears ($h$ in the observation $O$ and $l$ in the document $W_k$) while $|W_k|$ is the number of words in $W_k$ (i.e., $|\;|$ gives the set cardinality). More details can be found in [6].

### 2.2. Zero-Frequency Problem

$Pr(w_{k,i})$ was estimated from $c(w_{k,i})$, the number of times in which each word appears in a given document. By definition $c(h) = 0$ implies $Pr(w_{k,i}) = 0$ that is, the product $\prod_{j=1}^{m} Pr(o_j|W_k)$ is zero, whenever an observation $o_j$ is not present in the training set of the document $W_k$.

To overcome this problem, known as the zero-frequency problem [31], different approaches were proposed. Results presented in this paper were obtained with a closed vocabulary, where probabilities were smoothed according the shift-$\beta$ smoothing technique.

Details and a complete overview on different smoothing techniques and probability estimations can be found in [16].

### 2.3. Leave-One-Out Validation

Collecting data is a costly activity, and several validation methods are available to minimize the data collection costs while maximizing the information obtained from the data. Widely adopted is the leave-one-out (also known as Jackknifing) method [25]: a cross-validation method in which, for a complete data set with $N$ samples, one sample is used for testing and $N-1$ samples are used for training. This process iterates for a total of $N$ times, each time with different withheld datum and thus with different training and test sets.

## 3. Traceability Recovery in Industral Systems

With respect to software systems completely developed from scratch, such as those analyzed in [6], an industrial system may contain:

- Partially automatic-generated code;
- Totally automatic-generated code;
- COTS and reused code;
- External Architectures (e.g., ORBs, network infrastructures, etc.); and
- Design and implementation-level components.

In the following sub-section, each item will be analyzed, and relations to traceability issues established.

### 3.1. Partially Automatic-Generated Code

These are components and classes, often related to the GUI of the system, containing both automatically generated code (and identifiers) and code written by programmers. For example, identifiers and names of GUI widgets (labels, text fields, buttons, etc.) are automatically generated following precise naming conventions (e.g., `TButton1, TButton2, TTextField1`, etc.). Naming conventions may be customized; widget identifiers and names changed, from a property window of the GUI builder. However, programmers tend to give significant (and domain consistent) names only to a small fraction of automatically generated component elements, i.e. only to those they consider essential to understand the application while developing the system. Often default names dominate (particularly for labels, shapes and any visual component having a constant value and behavior) over programmer specified names.

Similar considerations may be applied to those visual components, encapsulating specific functionalities and dragged into user interfaces. Any kind of ”visual COTS”, like Microsoft ActiveX or OCX and Borland VCL, belongs to this category.

Consider a component implementing a DBMS connection and queries. The component may have a default-generated name (e.g., `Table1, Table2, Query1`) or a user-defined name; if the name is not consistent with the related requirements, traceability will be lost. Notice that, these components may encapsulate complex behaviors (update a database table, retrieve a page from the web, print a

PDF document, etc.). Thus, the more complex the encapsulated behavior, the more severe the traceability problem is.

To maximize traceability, these default-generated identifiers should be filtered out as a preliminary step. Assume, for example, two automatically generated windows, containing the same number and types of widgets, they will exhibit very similar, or identical, identifier frequency distribution, although they are associated to different domain concepts and requirements. Even worse, generated identifiers are likely to dominate over identifiers forced by programmers, thus reducing the traceability link recovery accuracy.

## 3.2. Totally automatic-generated code

These are components and classes automatically generated (e.g., components generating reports, created by report generators) without any need of coding; the programmer intervention, if any, is almost irrelevant (e.g., fill the printer spooler device property of the visual component).

Although these components and classes implement functionalities specified into high-level documents, traceability recovery is impossible. Suppose the requirement document contains the following sentence:

*"The system shall allow to print the list of customers from a particular town, specifying names, address. . . "*

Such a requirement is simply implemented by an automatic-generated report, containing only identifiers like `QRBand1`, `QRBand2`, `QRLabel1`, `QRDBText1`, `QRLabel2`, `QRDBText2`, etc.

To ensure traceability, a programmer manual intervention is required, either recording traceability links or assigning consistent names while using the report generator (e.g., the fields `QRDBText1` and `QRDBText2` should be mapped into `Customer_name`, `Customer_address`, etc.).

## 3.3. COTS and Reused Code

Usually, COTS source code is not available; the only traceable elements are the uses of COTS resources (e.g., classes, functions, etc.). Consistency between COTS resource names and high-level documents is essential. If COTS code is available, it is unlikely that COTS identifiers reflect the application domain concepts and high-level documents names. As for automatically generated code, manual intervention is required.

Reused components are equivalent to COTS, apart from the availability of source code. Traceability may be brought back to the same situation of COTS. The code may have been developed in previous projects related to different application domains; consistency between code identifiers and requirement terms is very unlikely. In other words, reused components and classes should be excluded from the traceability recovery process.

## 3.4. External Architectures

This category encompasses middleware (e.g., CORBA) and frameworks (e.g., Zope). External architectures are similar, at a different level of granularity, to the partially generated code. System components or system class hierarchies are in relation with other external hierarchies of classes. When an external architecture is exploited, there may be automatically generated code and classes (with or without programmer hooked code), classes partially generated, derived from the external hierarchies (with or without overloading) or reused from a framework.

A typical example is the use of an ORB in a distributed software system to guarantee interoperability between application objects and remote objects. A node of the distributed system may contain the following families of classes:

- Classes that implement portion of the CORBA architecture (reused classes). As for other reused code, the traceability link recovery is, in general, not possible;
- Automatic generated classes: these classes are generated by the IDL compiler, typically to implement data structures passed around by distributed objects. Since original variable names (as defined in the IDL specification files by programmer) are modified and several other variables are generated, there is no guarantee to recover traceability links; and
- Partially generated classes: these are, for example, the CORBA stubs and skeletons, containing methods and identifiers with names as defined by programmer in the IDL. These classes can be successfully traced to requirements specifying information communicated to/from remote systems (implemented using CORBA).

## 3.5. Design and Implementation-Level Components

These components are not referred into requirements. Therefore, there is no guarantee to ensure their traceability. Consider, for example, the splash screen or the about window of a GUI application. Usually, there is no requirement related to these classes. The motivation is grounded into the object-oriented development process itself. Requirements are well mapped into classes reflecting the domain-component view [13, 14] or the conceptual view of the system [10, 18]. At the design level, the class hierarchy is modified according to [13, 14] or [10, 18], refining domain object and adding details such as reused classes, human interaction classes, data management classes, and task management classes. These components and classes cannot be traced directly to high-level document; an intermediate step, tracing them into design may be required.

## 4. The Traceability Recovery Process

Figure 1 shows the process defined to recover traceability links between high-level documents and source code. The process accounts for the affecting factors identified in the previous section; it was instantiated in the context of traceability link recovery between functional requirements and the corresponding object-oriented code i.e., C++ in our case study.

The underlying approach assumes the *sane programmer* conjecture; it further assumes that, for each requirement, at least one traceability link is available (continuous lines in Figure 2).

The knowledge of existing traceability links bootstraps the recovery of remaining unknown links (dashed lines in Figure 2).

The recovery process can be thought of as composed by three main blocks, each one further decomposed in sub-blocks:

1. Requirements processing;
2. Code processing; and
3. Traceability map recovery by means of the Bayesian classifier explained in Section 2.

### 4.1. Requirements Processing

Each requirement is processed and the dictionary of its words, scored by word frequency, extracted. Stop words (e.g., articles) are discarded; successively words undergo a normalization step of morphological analysis (the Stemmer box in Figure 1). Plural and singular conjugated verbs, as well as synonyms, are brought back to the radix.

It is worth noting that, in general, this phase cannot be fully automated, and it is likely to remain semi-automatic; it is well known that word semantic may be context sensitive, and currently available natural language tools cannot fully disambiguate contextual semantic, nor deal with complex form representing metaphoric speech (e.g., …*the process should be killed*…).

### 4.2. Code Processing

For each class, a list of identifiers and the associated occurrences is extracted from both interface and implementation files. On the contrary of previous works [6, 4], comments are not discarded.

The association comments/classes is immediate if the system is organized so that for each class there is an `.h` file and a `.cpp` file; otherwise, heuristics need to be adopted: if a comment is not inside a class interface or inside a method implementation, it may be associated to the immediately following/preceding class/method.

Words are preliminary filtered by a stopper. Stop words removal is divided into two sub-phases; in the first sub-phase the stopper discards the following elements:

- The same words removed from requirements;
- Language reserved keywords and language predefined types; and
- Short identifiers, commonly used as cycle counters or array indexes (e.g., `x`, `y`, `j`, `k`).

It is worth noting that the phase requires human intervention: short words may convey relevant application domain information.

The first sub-phase is followed by the removals of automatic-generated identifiers. As highlighted in Section 3, they may confuse the traceability recovery. Pruning is performed automatically once the list of classes available in the RAD tool library is known; identifier will have a name generated following well-defined rules (e.g., for the class `TLabel` identifiers generated will have names like `TLabel1`, `TLabel2`, etc.).

Morphological analysis is subsequently applied to the remaining words (the *Normalizer* box in Figure 1). However, the applied analysis is slightly different from those performed on requirements. Plural and singular verb forms are handled; furthermore, method names handling events dispatched by the same object are brought back to a radix equal to the object name. For example, if there is a text field named `Address`, methods named `AddressClick`, `AddressChange`, etc. are brought back to the radix `Address`. This phase does not require human intervention when RAD tools are used; the method names have the object name as prefix, followed by the event name (the list of possible events is known).

Finally, as underlined in Section 3, classes, entirely generated, that cannot be traced into high-level documentation, are removed from the system under analysis. This step can be performed automatically once the tools used to develop the system are known.

Reused code and external architecture are processed as untraceable classes, and removed from further considerations. Unfortunately, given the granularity level, the process is more complex and more human-intensive with respect to automatically generated code. However, file names, comments, documentation (if available) or methods like those presented by Antoniol et al. in [7] may serve to facilitate the activity.

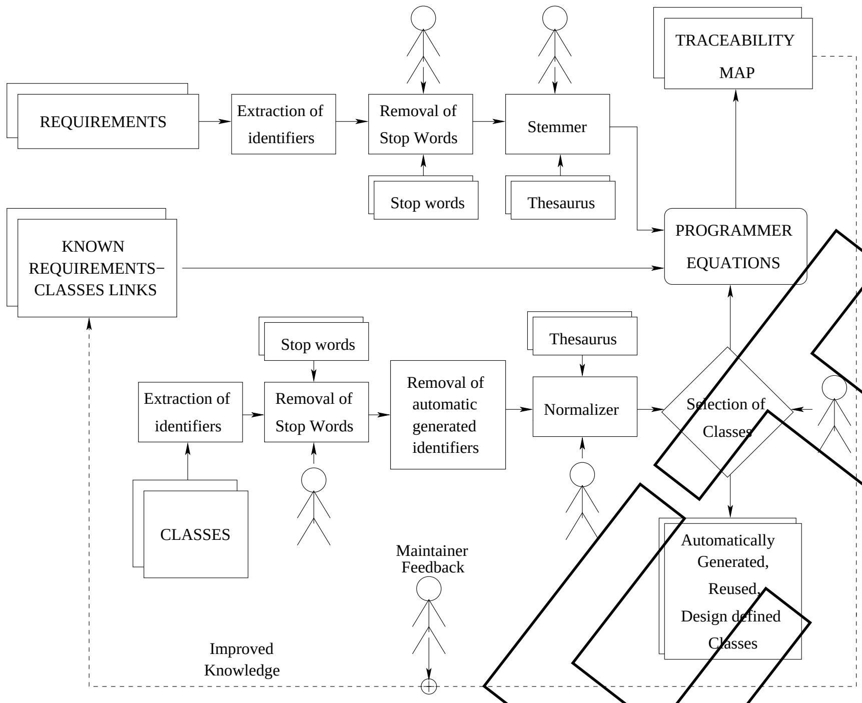

**Figure 1. The Traceability Recovery Process**

### 4.3. Traceability Map Recovery

Equations outlined in Section 2 are applied to the a-priori known links, and probabilities on training material (the filtered information) estimated. Subsequently, the Bayesian classifiers score traceability links with probability. In other words, it is very likely that the correct link shows up in the top-most positions.

New links are in turn used as new inputs, enriching the classifier training set; the re-trained classifier is then applied to the remaining class-requirement couples.

## 5. Tool Support

To automate the traceability recovery process, a number of tools have been developed:

1. A script to extract words and their occurrences from requirements, plus a script that parses the C++ source code and extracts the list of couples identifier/occurrence (keywords, language types and language's symbol are removed);
2. A stemmer that removes stop-words (such as articles, numbers, preposition, certain categories of verbs etc.) from requirements and code. The maintainer is left in charge (by modifying the stop word list) of the final decision whether or not a word should be removed;
3. A morphological analyzer, based on a thesaurus produced by the maintainer from the dictionary of all the possible terms of the requirements/code;
4. A script that, given the names of library classes used by the RAD or present in the middleware, removes all automatic-generated identifiers from the code; and
5. Finally, a tool that implements the estimation of probabilities $Pr(o_j|w_{k,i})$, $Pr(w_{k,i})$ with the shift-$\beta$ smoothing and closed vocabulary.

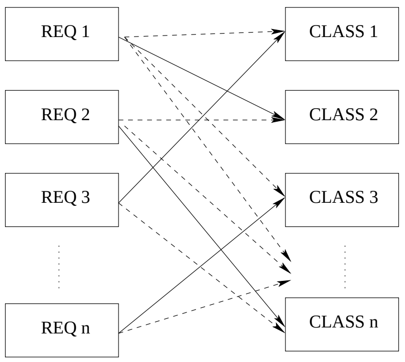

**Figure 2. An Example of Traceability Links**

## 6. Case Study

The process proposed in Section 4 and the tools developed were applied to an industrial system: *Transient Meter* [15]. *Transient Meter* is a distributed measurement system for power quality monitoring developed at the Measurement Laboratory of University of Sannio, and currently under beta-test by the Italian power agency. The system is composed by nodes placed near sites to monitor (e.g., generators, motors, industrial plants), and connected with it by a trigger circuit (that detects the power disturbance) and a digital acquisition board. These nodes detect power transients, processes them (performing classification and measurement) and then sends them to a central node, that retrieves data from distributed nodes and stores it into a database.

The central node, on which code we performed our traceability recovery experiment, performs other activities such as system startup, configuration monitoring, waveform processing (e.g., FFT), and visualization.

The system suffered of almost all traceability problems outlined in Section 3:

1. The system was developed using a RAD development tool ($BorlandC++Builder^{TM}$), therefore it contains both totally-automatic generated classes (e.g. classes generated by report generators) and classes containing some identifiers automatically generated (e.g. user interface classes);

2. The system reused existing class hierarchies: one to read and write data from/to `.wav` files, another one for signal processing. Moreover, COTS components were used for handling complex signal visualizations, to implement some user interface widgets typical of a measurement instrument (led, switches, seven-segment displays, etc.), and to handle Fourier transforms; and

3. Finally, the communication between the central node and the distributed nodes was implemented by a CORBA architecture.

The *Source Requirements Specification* document contains 13 requirements, and the source code (limited to the only central node) consists of 36 classes (for a total of about 15 KLOCs), of which three were automatically generated by the RAD tool, seven are reused classes, four are design-level classes, and three classes belongs to the CORBA architecture. Other classes generated by the IDL compiler to define data structures exchanged between distributed objects were disregarded; this was possible since these classes had names with a well-known postfix: `_var`, `_forany`.

## 7. Experimental Results

Traceability links recovery was performed on *Transient Meter* according to the process described in Section 4, and applying a leave-one-out approach as in [6]. Results are reported in tables showing the precision and the recall [17] for different sizes of the training set (i.e., one, two and tree known links for each requirement. *Precision* is the ratio of the number of relevant documents retrieved over the total number of documents retrieved. *Recall* is the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in the "database". Precision and recall were computed for first, second and third position in the scoring (i.e. the best one, two and three classes associated to a requirement). Each data point in the tables corresponds to 100 random experiments; the number of experiment replications was chosen to guarantee a standard deviation of precision and recall below 3%.

A comparison baseline, corresponding to the process presented in [6] was first defined. In other words, the traceability links were recovered on information obtained by removing stop words and applying the thesaurus based morphological analysis. As shown by results summarized in Table 1 precision and recall were very low compared to those presented in [6].

| **Score:** | | **Best 1** | **Best 2** | **Best 3** |
|---|---|---|---|---|
| 1 Class | Precision (%): | 6.5 | 7.8 | 8.9 |
| Training | Recall (%): | 6.5 | 15.6 | 26.1 |
| 2 Classes | Precision (%): | 4.1 | 8.1 | 9.3 |
| Training | Recall (%): | 4.1 | 16.2 | 27.8 |
| 3 Classes | Precision (%): | 6.1 | 8.2 | 10.9 |
| Training | Recall (%): | 6.1 | 16.4 | 32.7 |

**Table 1. Results after stop-words pruning and normalization of identifiers.**

Differences may be explained in terms of the adopted software development approaches. The *Transient Meter* prototype was developed with RAD IDE, COTS, communication middleware, while *Albergate* [6] was coded from scratch without reusing components but a relational database.

Data reported in Table 1 are puzzling: adding information decrease traceability. Training the model with three classes lowered accuracy. This phenomenon happens when the class added has few (or no) common identifiers with the other classes traced to the given requirement. Such a class adds "noise" rather than useful information causing classes of the *test set* to be easily associated with a wrong requirement.

We also experienced that, performing morphological analysis, results were not improved ameliorate the results.

However, further experiments demonstrated that this step could not be removed from the process without negatively affecting final results. Moreover, the step helped to eliminate the decrease of performance when increasing the size of the *training set*. In fact, normalizing words and class identifiers associates classes whose identifiers (attributes, methods or comments) share common radixes.

However, in presence of automatically generated code and COTS, the normalization tends to add noise, confounding links; automatically generated code and COTS need to be removed to benefit from this step.

Successively, automatic-generated identifiers were pruned and results, as shown in Table 2, tends, in general, to be slightly better than those of Table 1.

| **Score:** | | **Best 1** | **Best 2** | **Best 3** |
|---|---|---|---|---|
| 1 Class | Precision (%): | 5.2 | 9.2 | 10.2 |
| Training | Recall (%): | 5.2 | 18.4 | 30.5 |
| 2 Classes | Precision (%): | 1.2 | 7.5 | 10.7 |
| Training | Recall (%): | 1.2 | 15.0 | 32.1 |
| 3 Classes | Precision (%): | 2.1 | 9.0 | 13.4 |
| Training | Recall (%): | 2.1 | 17.9 | 40.1 |

**Table 2. Results after removal of automatic-generated identifiers.**

Classes, that cannot be effectively traced into requirements, as explained in Section 3, were removed in the next step.

In the current case study, these classes consist of one splash screen, a window to display aggregated data from the database, and a class for report printing. Results are shown in Table 3. It is worth noting that, while results tends, in general to be better, first position precision and recall tend to be very low, in particular when training with two or three classes. As shown in the final step, this is due to design-level classes that tend to be erroneously coupled with some requirements, and these classes identifiers "dominate" once some others (automatic-generated classes, reused, classes, etc.) were removed.

| **Score:** | | **Best 1** | **Best 2** | **Best 3** |
|---|---|---|---|---|
| 1 Class | Precision (%): | 6.3 | 11.2 | 11.6 |
| Training | Recall (%): | 6.3 | 22.5 | 34.8 |
| 2 Classes | Precision (%): | 1.7 | 12.0 | 15.1 |
| Training | Recall (%): | 1.7 | 24.0 | 45.4 |
| 3 Classes | Precision (%): | 2.4 | 11.6 | 15.6 |
| Training | Recall (%): | 2.4 | 23.3 | 46.7 |

**Table 3. Results after removal of automatic-generated classes.**

We further discovered that, in *Transient Meter*, reused classes have no way to be traced into requirements. Thus, seven classes (two for handling wave files, five for signal processing), all mapped to a single requirement, were excluded prior to compute new results. Moreover, classes belonging to the CORBA architecture (three classes) except the stub, were removed. Results obtained are shown in Table 4. It is worth noting that even though the precision and recall for the first position tend to be 0, results for second and third positions were considerably ameliorated.

| **Score:** | | **Best 1** | **Best 2** | **Best 3** |
|---|---|---|---|---|
| 1 Class | Precision (%): | 6.7 | 12.9 | 14.7 |
| Training | Recall (%): | 6.7 | 25.9 | 44.2 |
| 2 Classes | Precision (%): | 0.0 | 24.0 | 21.4 |
| Training | Recall (%): | 0.0 | 48.0 | 64.2 |
| 3 Classes | Precision (%): | 0.0 | 24.0 | 23.0 |
| Training | Recall (%): | 0.0 | 47.9 | 68.9 |

**Table 4. Results after removal of reused classes and CORBA classes.**

The last step aimed to remove low-level design classes, that cannot be directly mapped to requirements. A total of four classes, three handling data structures and one implementing an adapter to reused classes, were removed. Final results (in which the remaining 19 of 36 classes are considered) are shown in Table 5. As shown in the table, the last step also removed the low precision and recall for first position. A detailed investigation revealed that the phenomenon was caused by some design-level classes, and also by some automatic-generated classes, and that the improvement resulted visible only after removal of both types of classes (See *Best 1* column in Tables 3, 4 and 5).

| **Score:** | | **Best 1** | **Best 2** | **Best 3** |
|---|---|---|---|---|
| 1 Class | Precision (%): | 26.1 | 24.1 | 19.8 |
| Training | Recall (%): | 26.1 | 48.2 | 59.6 |
| 2 Classes | Precision (%): | 59.6 | 38.0 | 27.6 |
| Training | Recall (%): | 59.6 | 76.0 | 82.8 |
| 3 Classes | Precision (%): | 71.1 | 45.8 | 31.1 |
| Training | Recall (%): | 71.1 | 91.5 | 93.3 |

**Table 5. Results after removal of design-level classes.**

## 8. Lesson Learned

It is not unlikely that industrial software contains COTS, reused code, communication middleware and, more generally, components that are difficult or even impossible to trace into requirements. The detailed analysis of our case study revealed that several of such components were present. As for automatically generated GUI, there is no way to automatically trace those low-level artifacts into high-level documentation. Even worse, we discovered that traceability accuracy was dramatically lowered by the presence of non-traceable artifacts.

When using RAD IDE environment, reused code, external architectures/middleware, programmers tend to assign meaningful names only to a fraction of the names, as a consequence non-domain specific names may dominate over domain related names thus confusing the traceability recovery process. Non-traceable elements should be removed from the analysis; in the case of automatically generated code we adopted an heuristic to discard classes that, after pruning automatic generated identifiers, exhibited a list of identifiers empty or below a fixed threshold. However, in this step as in other phases such as morphological analysis. It is very likely that human intervention will always be required. Consider, for example, *Transient Meter*: as a rule, short words should be removed. However, the system measures quantities related to the network distributing the power, thus the letters A, B, C indicate phases of a tri-phase power line and must not be removed.

While performing the traceability recovery process, we discovered that comments are a valuable source of information, thus on the contrary of previous work [4, 6] comments were exploited to recover traceability links. Clearly, this required a coding standard to help associating comments with classes and methods. In agreement with previous contributions [4, 6], text normalization was fundamental; however, text normalization have to be applied taking into account the identified affecting factors, and thus following the proposed process. By simply applying text normalization, very poor results were obtained.

We also observed, that, once the process is applied, enriching the training set increases the precision/recall of subsequent steps. This fact is very relevant to reduce human intervention since, as for *Transient Meter*, without applying the proposed process, adding further information will not results in an improvement. This means that a compromise should be pursued between code and requirements preprocessing effort and traceability recovery effort.

## 9. Related Work

The issue of recovering traceability links between code and free text documentation, in the authors' knowledge, is not yet well understood and investigated, and very few contributions were published in the past 20 years. A number of related papers are in the area of impact analysis. For example, Turver and Munro [26] assumed the existence of some form of ripple propagation graph describing relations between software artifacts, including both code and documentation, and focused on the prediction of the effects of a maintenance change request, not only on the source code but also on the specification and design documents.

Boldyreff et al. [9], presented a method for the identification of traceability links of high-level domain concepts, using all available maintenance information and starting from a different view of traceability respect to IEEE definition [1].

Antoniol et al. [5] presented a method to establish and maintain traceability links between code and free text documents. The method exploits probabilistic information retrieval techniques to estimate a language model for each document or document section, and applies Bayesian classification to score the sequence of mnemonics extracted from a selected area of code against the language models. The same method was applied in [3], to recover traceability links between the functional requirement and the Java source code, extending and validating the previous results on a more complex and difficult case study. The investigation was then extended in [2] to vector space models, to compare different models families and to assess the relative influence of affecting factors.

The case study, the equations, the approach and the process proposed in this paper differ in several aspects from those of papers mentioned above. More commonalities can be found with [6]. This paper is focused on a taxonomy of affecting factors we identified, affecting factors that are likely to be present in any industrial system. We demonstrated that by removing affecting factor traceability recovery accuracy was considerably improved.

## 10. Conclusions

This paper presented an approach and a process to filter information gathered form low level and high level software artifacts to recover traceability links. The process aims to remove factors affecting traceability links recovery. The identified affecting factors stem from industrial practice adopted to reduce time to market (COTS, middleware, reused and generated code).

The process was applied to recover traceability links in a system developed with RAD IDE, code generators, and incorporating middleware and reused code. Results obtained following the process and using the Bayesian classifier proposed in [6] were encouraging, and comparable with those obtained in [6] on a much simpler code (i.e., Java, no RAD, COTS, generated or reused code). Intermediate results showed how different factors, discussed in Section 3, influenced performances. The filtering process is not tied to Bayesian classification or equations 4 and 5, thus any traceability recovery method (even the `grep`) will benefit from its application.

Future works will be devoted to apply the process to different case studies, and to recover a complete traceability map through the documents produced at different phases of the software life-cycle (i.e., use-cases, design documents, testing documents, etc.).

## References

[1] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Computer Society Press, 1991.

[2] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of IEEE International Conference on Software Maintenance*, San Jose, CA, 2000. IEEE Computer Society Press.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Tracing object-oriented code into functional requirements. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 227–230. Limerick, Ireland, June 2000.

[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, (To appear).

[5] G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. Recovering code to documentation links in oo systems. *Proc. of the Working Conference on Reverse Engineering*, pages 136–144, Oct 1999.

[6] G. Antoniol, G. Casazza, and A. Cimitile. Traceability recovery by modeling programming behavior. In *Proceedings of IEEE Working Conference on Reverse Engineering*, Brisbane, Australia, 2000. IEEE Computer Society Press.

[7] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. A method to re-organize legacy systems via concept analysis. In *Proceedings of the IEEE International Workshop on Program Comprehension*, Toronto, ON, Canada, May 2001. IEEE Press.

[8] T. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, Jul 1989.

[9] C. Boldyreff, E. Burd, R. M. Hather, M. Munro, and E. Younger. Greater understanding through maintainer driven traceability. In *Proceedings of the International Workshop in Program Comprehension*, Germany, March 1996. IEEE Press.

[10] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unifi ed Modeling Language User Guide*. Addison-Wesley Publishing Company, 1998.

[11] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[12] E. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[13] P. Coad and E. Yourdon. *Object Oriented Design*. Yourdon Press Computing Series, 1991.

[14] P. Coad and E. Yurdon. *Object Oriented Analysis - Second edition*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[15] P. Daponte, M. Di Penta, and G. Mercurio. Transient meter: A distributed measurement system for power quality monitoring. In *Ninth International Conference on Harmonics and Quality of Power*, pages 1017–1022, Orlando, Florida, October 2000.

[16] R. De Mori. *Spoken dialogues with computers*. Academic Press, Inc., Orlando, Florida 32887, 1998.

[17] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[18] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unifi ed Software Development Process*. Addison-Wesley Publishing Company, 1999.

[19] S. Letovsky. *Cognitive Processes in Program Comprehension: First Workshop. E. Soloway and S. Iyengar eds.* Ablex Publisher, 1986.

[20] E. Merlo, I. McAdam, and R. De Mori. Source code informal information analysis using connectionist models. In *Proceedings of the International Joint Conference on Artifi -cial Intelligence*, pages 1339–1344, Chambery France, Sept 1993.

[21] N. Pennington. *Comprehension Strategies in Programming. In: Empirical Studies of Programmers: Second Workshop. G.M. Olsen S. Sheppard S. Soloway eds.* Ablex Publisher Nordwood NJ, Englewood Cliffs, NJ, 1987.

[22] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[23] B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behaviour: A model and experimental results. *IJCIS*, 8(3):219–238, Mar 1979.

[24] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1994.

[25] M. Stone. Cross-validatory choice and assesment of statistical predictions (with discussion). *Journal of the Royal Statistical Society B*, 36:111–147, 1974.

[26] R. J. Turver and M. Munro. An early impact analysis technique for software maintenance. *Journal of Software Maintenance - Research and Practice*, 6(1):35–52, 1994.

[27] I. Vessey. Expertise in debbugging computer programs: A process analysis. *IJMMS*, 23:459–494, 1985.

[28] A. Von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. In *Proceedings of IEEE Workshop on Program Comprehension*, pages 78–86, Capri Italy, 1993. IEEE Comp. Soc. Press.

[29] A. Von Mayrhauser and A. M. Vans. Dynamic code cognition behaviours for large scale code. In *Proceedings of IEEE Workshop on Program Comprehension*, pages 74–81, Washington DC USA, 1994. IEEE Comp. Soc. Press.

[30] A. Von Mayrhauser and A. M. Vans. Identifi cation of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, 1996.

[31] I. H. Witten and T. C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Trans. Inform. Theory*, IT-37(4):1085–1094, 1991.