# Design-code traceability for object-oriented systems

Giulio Antoniol [a], Bruno Caprile [b], Alessandra Potrich [b] and Paolo Tonella [b]

[a] *Research and Technology Department, University of Sannio, Faculty of Engineering, P.zo Bosco Lucarelli, 82100 Benevento, Italy*
E-mail: antoniol@ieee.org

[b] *ITC-Irst, Istituto per la Ricerca Scientifica e Tecnologica, Via alla cascata, 38050 Povo (Trento), Italy*
E-mail: {caprile,potrich,tonella}@itc.it

Traceability is a key issue to ensure consistency among software artifacts of subsequent phases of the development cycle. However, few works have so far addressed the theme of tracing object oriented (OO) design into its implementation and evolving it. This paper presents an approach to checking the compliance of OO design with respect to source code and support its evolution. The process works on design artifacts expressed in the OMT (Object Modeling Technique) notation and accepts C++ source code. It recovers an "as is" design from the code, compares the recovered design with the actual design and helps the user to deal with inconsistencies. The recovery process exploits the edit distance computation and the maximum match algorithm to determine traceability links between design and code. The output is a similarity measure associated to design-code class pairs, which can be classified as matched and unmatched by means of a maximum likelihood threshold. A graphic display of the design with different green levels associated to different levels of match and red for the unmatched classes is provided as a support to update the design and improve its traceability to the code.

## 1. Introduction

Software systems are developed following phased processes in which software engineering complexities are tackled by means of subsequent refinement activities. Requirement analysis, design and coding are phases that are present in almost any software development process. A phased process however does not automatically help to trace how requirements evolve into design and design into code.

Design documents are an important source of information, especially when the system enters the maintenance phase [Baxter and Pidgeon 1997]. However, maintaining consistency between software artifacts is a costly and tedious activity frequently sacrificed during development and maintenance due to market pressure. This often causes design to quickly become obsolete with respect to source code.

The verification of the design-code compliance is the basic step to produce an updated version of the design. Even if reverse engineering tools can extract design representations from the code, it is preferable to evolve existing design so that it matches the code. In fact, designs produced by users are usually richer than those extracted automatically, since they include context and high level semantic information. Further-

more, in reverse engineered designs multiple semantics are candidate explanations for the same piece of code (e.g., in C++ a pointer may be used to implement an association as well as an aggregation). That's why evolved designs are considered to be of higher quality, provided that traceability with code is maintained.

The activity of checking the compliance and evolving the design can be greatly assisted by automatic tools. Few approaches and systems to monitor the implementation faithfulness to its design have been proposed in the literature [Luckham *et al.* 1987; Meyers *et al.* 1993; Murphy *et al.* 1995; Sefika *et al.* 1996]. In [Luckham *et al.* 1987], a language for annotating Ada programs is defined. Meyers *et al.* [1993] designed and implemented a language, CCEL, based on assertions to express constraints on the structure and style of object oriented programs implemented in C++. Murphy *et al.* [1995] developed an approach, called the *software reflexion models*, in which the user provides a high-level model of the system and a map stating how entities in the high-level model should be associated to those found in the source code. Sefika *et al.* [1996] proposed a hybrid approach which, by integrating logic-based static and dynamic visualization, helps determining design-implementation congruence at various levels of abstraction, from coding guidelines to architectural models such as design patterns [Gamma *et al.* 1995] and connectors [Garlan and Shaw 1996], to design principles like low coupling and high cohesion.

An in field study of traceability for a system at Ericsson is described in [Lindvall and Sandahl 1996]. The reported experience suggests that by emphasizing traceability as a quality factor from the very beginning, the documentation will be clearer and more consistent. However, tracing items with no tool support or in models partially inconsistent and underdocumented requires significant effort.

This paper presents an approach to checking the compliance and evolve OO design with respect to source code, extending the preliminary work described in [Antoniol *et al.* 1999]. The process operates on design artifacts expressed with the Object Modeling Technique (OMT) [Rumbaugh *et al.* 1991] notation and accepts C++ source code. Both design and code are represented using a custom OO design description language, the Abstract Object Language (AOL). The process recovers an "as is" design from the code in AOL, compares the recovered design with the actual AOL design and helps the user to deal with inconsistencies by providing a similarity measure for the matched classes and pointing out the unmatched ones. This activity was partially funded by Sodalia SpA[1] under the DEMOS 2 project, aiming at estimating software size and complexity, and improving its quality.

Bunge's ontology [Bunge 1977, 1979] has been taken as the base conceptual framework to define the *similarity* criterion. An object is viewed as an individual which possesses properties. Comparing individuals for similarity translates into checking the similarity of the individuals' properties. When instantiated in the context of OO design to code traceability, individuals become classes, while properties are mapped into class attributes (fields and methods). A similarity measure between the names of

[1] Sodalia is one of the leading telecommunication software companies in Italy.

the attributes is proposed, based on the edit distance between strings, and the maximum match algorithm [Cormen *et al.* 1990] is applied to extract the best matching attribute pairs. A summary similarity between the compared classes is then computed as the average similarity measure of their attributes. The desired traceability links between design and code are retrieved by iterating such computation over each pair of classes and applying the maximum match algorithm at the class level. A further step can be performed to separate matched classes from unmatched ones, by means of a maximum likelihood classifier [Duda and Hart 1973].

Traceability of the relations can then be investigated for the matched classes. Since the design represents an abstraction of the implementation, relations between classes in the design are expected to be all present in the code, while additional relations in the code can be regarded as implementation details. Thus the deletion of a relation from the design is considered a traceability fault, and is signalled to the user, while the addition of new relations in the code does not necessarily imply the need of a design update.

Another traceability index is provided by the dictionary of words used to build compound identifiers. The same names and acronyms as used in the design should be found in the code, since they are the only means to ensure traceability. By segmenting compound identifiers into the composing words, a design/code dictionary can be constructed which allows the update of both design and code to a standardized and traceable set of accepted and recognized terms.

The proposed approach has been experimented on industrial design and code. Support tools have been developed to extract the "as is" design from source code, to match design into code and finally for result visualization. To communicate and discuss our findings with project managers and programmers a pair-difference coloring technique was adopted. A colored class diagram summarizes design-code traceability relations by assigning different colors respectively to perfectly matched information, information present in the design but absent in the code (unmatched items) and information partially matched.

Context information in the design and automatically generated code can decrease traceability. Classes from other components can be included for clarity in the design of the current one, without being associated to developed code, while on the other side classes generated, e.g., by Graphical User Interface (GUI) builders could be undocumented in the design. Information in this category is tagged as unmatched by the proposed approach, and the one related to the design is displayed in a different color (red), so that it is not confused with that requiring a design update (light green). Unmatched information in the design has to be retained if it helps understanding the whole context, but it should be marked differently from the rest of the diagram, as suggested by the resulting colored class diagram.

For all the analyzed components a summary with the average match level and the number of unmatched classes will be given. Relations deleted from the design when moving to the code will also be outlined. Furthermore, one case study component will be investigated in more detail and its traceability links will be considered from the class

level down to the individual fields and methods – thereby showing the multiplicity of granularity of the proposed approach. Finally, traceability will be considered in terms of the dictionary used to build compound identifiers in the design and in the code.

In section 2, a formal framework for design-code traceability, based on the Bunge's ontology is presented, followed by a detailed description of its instantiation for design-code traceability check. The adoption of a maximum likelihood classifier is introduced with the purpose of determining matched and unmatched class pairs. The traceability of the relations is then considered, and an analysis of the dictionary used in design and code is presented. In this section some issues related to handling context information are also discussed, and finally a visualization technique to show the results in an intuitive form is considered. Section 3 is devoted to an experimental validation of the approach on an industrial system for which both design and code are available. In section 4, our approach is compared with related work in design-code compliance verification. Finally, in section 5 conclusions are drawn.

## 2. Evolving object-oriented design

Each software artifact along the software development cycle should be the "refinement" of the artifacts of the previous phase. It should be consistent with the previous artifacts and it should be possible to trace information along the phased development process.

### *2.1. Design-code match*

Bunge's ontology has been a source of inspiration in the OO domain. According to this ontology objects can be viewed as *substantial individuals* which possess *properties*. Chidamber and Kemerer [1994] proposed a representation of *substantial individuals*, objects, as a finite collection of properties:

$$X = \langle x, P(x) \rangle, \tag{1}$$

where the object $X$, is identified by its unique identifier, $x$, and $P(x)$ is its finite collection of properties.

In general, two objects $X$ and $Y$ may possess different properties. Thus, a preliminary step in the definition of a similarity measure between them is the introduction of a map $m$ between a subset of the properties of $X$ and a subset of the properties of $Y$, to be considered as matched properties. Then the remaining properties from $P(x)$ and $P(y)$ are unmatched properties, respectively, of $X$ and $Y$:

$$m : P(x) \to P(y), \tag{2}$$

$$\text{Unmatched}(X) = P(x) - \text{Dom}(m), \tag{3}$$

$$\text{Unmatched}(Y) = P(y) - \text{Ran}(m), \tag{4}$$

where $m$ is an injective function from $P(x)$ to $P(y)$, Dom is the domain and Ran the range. Unmatched properties of $X$ are those not in the domain of $m$, while unmatched properties of $Y$ are those not in the range of $m$.

Given a measure of similarity, $s(p, q)$, between the matched properties $p \in P(x)$ and $q \in P(y)$ of two different objects $X$ and $Y$, an overall similarity measure between the objects can be obtained by applying a suitable average operator as, for example, the arithmetic average:

$$s(X, Y) = \frac{1}{|\text{Dom}(m)|} \sum_{p \in \text{Dom}(m)} s\big(p, m(p)\big). \tag{5}$$

An overall picture of the similarity between $X$ and $Y$ is therefore given by the sets of unmatched properties (Unmatched($X$), Unmatched($Y$)) and by the average similarity measure between matched properties ($s(X, Y)$). A more detailed information can be obtained by the individual similarity measures for the matched pairs of properties ($s(p, q)$, $p \in \text{Dom}(m)$, $q = m(p)$).

The main entities present in an OO design, which must be reflected and implemented in the corresponding code, are classes, objects and relations. Our approach works on designs represented with the OMT notation. When tracing an OO design into C++ code, among the three models used in OMT to represent an OO software system (class diagram, functional model, dynamic model), we focused on the class diagram. The class diagram is usually the first to be developed, common to many other OO methodologies and notations (e.g., Booch and Unified Modeling Language) and it is the only one that describes the system using specifically OO concepts.

When instantiating the above notion of similarity between objects to trace OO design into code, classes have to be considered as the basic entities, whose properties are the class attributes (both fields and methods). Given a pair of classes for which a similarity measure has to be determined, the similarities of the contained attributes (properties) have to be computed first. For such a purpose we propose to consider the names of the attributes, prefixed with class scope, as strings, and to compute the complemented edit distance [Cormen *et al.* 1990] between such strings:

$$s(a_d, a_c) = \frac{1 - d(a_d, a_c)}{|a_d| + |a_c|}, \tag{6}$$

where $a_d$ and $a_c$ are the qualified names of an attribute from the design and from the code, respectively, while $d$ is the edit distance. Since the upper bound for the edit distance is the sum of the lengths of the strings ($|a_d| + |a_c|$), the above similarity measure is between 0 and 1, being 0 when the two strings $a_d$ and $a_c$ have no character in common, and 1 when they coincide. After computing the similarity between each pair of attributes, the match function can be inferred by applying the maximum match algorithm [Cormen *et al.* 1990] to the bipartite graph in which nodes are, respectively, attributes from the design and from the code, connected by edges that are weighted with the similarity measures. The edges computed by the algorithm as those giving the maximum match define the desired match function. A further outcome of this algorithm

is the set of unmatched attributes in the design and in the code. Then an average similarity measure can be computed for the two classes, as the arithmetic average of the similarity measures in the edges selected by the maximum match algorithm.

By repeating the above procedure for each pair of classes it is possible to determine their respective average similarity measure. To determine the correspondence between the design and the code it is possible to exploit the maximum match algorithm again. In this case the nodes in the bipartite graph are, respectively, classes from the design and from the code, while edges are weighted with the average similarity measures. The edges extracted by the algorithm represent the traceability links between the design and the code. Each link is weighted with a similarity measure. In addition, an initial set of unmatched classes is determined as those having no traceability link attached.

### *2.2. Classification of matched and unmatched class pairs*

The presence of a traceability link between a class in the design and a class in the code is not sufficient to state that a match occurs. In fact, the similarity measure associated to the link may be very low, and the edge in the bipartite graph could have been selected by the maximum match algorithm only as an effect of maximizing the total match measure.

Therefore, the links connecting matched classes from design and code have to be classified to distinguish truly matched classes from unmatched ones. The use of a maximum likelihood classifier [Duda and Hart 1973] is thus proposed, giving a threshold which separates low similarity class pairs, to be considered unmatched, from high similarity matched classes.

When a similarity threshold is adopted, two kinds of errors can occur. The first error is the classification of truly matched classes as unmatched. The parameter which accounts for this error is the *recall*, computed as the ratio of correctly classified class pairs over the total number of truly matched class pairs. If recall is 1 no matched class is missed by the classifier. The second error is the classification of an unmatched class pair as matched. The parameter accounting for it is the *precision*, computed as the ratio of correctly classified class pairs over the total number of classes classified as matched. If precision is 1 no unmatched class is classified as matched.

The maximum likelihood classifier is the one which minimizes the sum of the two errors. While each error can be individually minimized in a trivial way (recall tends to 1 as the threshold is arbitrarily decreased, while precision tends to 1 when the threshold increases), the maximum likelihood classifier gives the threshold for which the likelihood of *both* errors is minimum.

The computation of the maximum likelihood threshold requires that a set of class pairs is correctly labelled as matched or unmatched. For each of the two categories the shape of the probability density has to be estimated from the frequency, and the intersection of the two curves gives the threshold. Probability densities can be estimated by assuming a Gaussian distribution and determining mean and variance, or

more generally by using raw frequency data or smoothed/fitted data as a substitute of the true densities.

The accuracy of the classifier is then evaluated on a test set, different from the one used to compute the threshold. In cases in which few examples are available, the evaluation can be conducted with a *cross validation* technique. Each component in turn is considered as a test case, while the remaining components are used to determine the threshold. A repetition of the test procedure is thus possible by changing the test component. Average performance and robustness can then be assessed on a wider base than a single test case.

### *2.3. Relations traceability*

In the sections above, traceability has been considered at the level of the basic design entities – classes and attributes. The concept can be expanded further by requiring that traceability holds also at the level of the *relationships* among classes. In particular, if we consider a class in the design and the corresponding class in the code, the relations of generalization, association and aggregation present in the design should be reflected in the code. Since the code is the implementation of the design, a relation appearing in the code with no counterpart in the design can be considered a detail that is ignored when abstracted to the design level. On the contrary, all design relations are expected to be found in the code.

The traceability check procedure for the relations uses the following strategy. For each relation in the design, if the two connected classes are matched by two corresponding classes in the code, a relation of the same type is looked for between the two matched classes in the code. Relations between class pairs unmatched in the code are not considered.

### *2.4. Dictionary traceability*

If implicit traceability is adopted when the design is refined into the code, names play a very important role. In fact, since no explicit traceability link is built, the only means to map a design item into a code item is by analyzing its name. It is therefore crucial that the designer and the programmer share a common dictionary of words and abbreviations.

Dictionary traceability from design to code is achieved when the dictionary used by the designer is the same as used by the programmer. Extensions in the code are allowed, since new words may be required by the implementative details, but replacements with synonyms or different abbreviations are indicators of poor traceability. Furthermore the absence in the code dictionary of a design term may indicate the loss of some concept or the degradation of the identifier self-documentation ability.

The first step to check dictionary traceability consists in the extraction of lexica. To this aim, a module able to semi-automatically segment identifiers against a given dictionary was developed. The procedure runs as follows: the dictionary is initially empty, and identifiers are read from a design/code component. Whenever the case

occurs that segmentation of an identifier cannot be completely achieved with the given dictionary, the program asks the operator for help. By visually inspecting the (partial) results of segmentation, the operator tries to figure out which strings need to be included in the dictionary in order to complete the segmentation. Such strings are then added to the dictionary, and the process is reiterated until all the identifiers are completely segmented. The result of the procedure is a collection of words necessary and sufficient to segment all the identifiers contained in the input component, i.e., the (design or code) *component dictionary*.

The second step consists in the manual labeling, according to their type, of all the words contained in each component dictionary. To this end, 9 different types of words have been preliminarly identified, distinguishing, for example, acronyms from English forms, from words contractions, and so on. In section 3.6, the possibility to trace the dictionaries extracted from the design into those extracted from the code will be discussed for the components in the test suite.

### *2.5. Handling context information*

Context information is often included in the class diagram, while extra information can be in the code due to automatic generation. When tracing design into code such information is classified as unmatched. On the side of the design, the typical context information is:

- Classes from other user components, necessary to better understand the context of the current component.
- Classes from libraries, included to represent subclassing of, or associations with, current component classes.
- Environment components interacting with the user ones.

Classes in the code but not in the design have typically the following origins:

- COTS: the use of Components Off The Shelf may introduce classes in source code which are not modeled in design.
- Automatically Generated Code: produced by code generation environments.
- Middleware: middleware software layers, for example CORBA, implementing distributed computing, introduce new classes in the code as a result of the process of stub generation.
- UI components: components implementing User Interfaces (UI) are likely to present in combination many of the previous features. In fact, often they are generated automatically through GUI builders and usually make heavy use of libraries.
- Test code: drivers or stubs used to test the component in isolation.

The proposed approach to design-code traceability allows the identification of unmatched classes, as those for which no traceability link is built by the maximum match

algorithm, or those which are classified as unmatched by the maximum likelihood classifier. Unmatched classes in the design can be retained for clarity, even if a good practice would suggest to mark them and make them recognizable as context information.

### *2.6. Difference visualization*

To highlight commonalities and differences, *pair-difference coloring*, a technique which employs different colors to contrast pairs of versions of the same information, was adopted. This technique has been previously used to represent changes in source code between two different versions of a software product [Holt and Pak 1996]. Common and different parts of the two versions are assigned different colors. Colors have also been used in software visualization, to associate time information distinguishing recent from older changes in source code [Ball and Eick 1996].

Evolution of the design of a software component can be supported by adopting such coloring techniques. We propose to modify the background color of the classes in the design so that those unmatched by the code are red, while those with a perfect match (similarity equal to 1) are green. Intermediate colors from green to yellow can be used for intermediate similarity levels. The programmer charged to update the design to improve its traceability with the code can prioritize the interventions. Classes with colors close to the green require few modifications in the names of the attributes, while yellow classes may require deeper investigations to understand why the best matched attributes in the code are so different. Red classes are not matched at all, thus the question is if they are context information to be preserved or if they have to be removed from the design.

In addition the programmer can go into a deeper detail and analyze the match between attributes (fields and methods) for a class of interest in the design. The traceability of the relations is also given, as the list of inter-class relations that are specified in the design but are not implemented in the code. This information is provided textually.

At present, a prototype of the difference visualization technique is implemented on top of OOD (Object-Oriented Designer)[2], a public domain design tool. The output of the tool to the printer is filtered by a program which uses the match levels determined by traceability analysis to add colors to the classes in the design. An example of the resulting plot will be given in the next section.

## 3. Experimental results

The whole design to code traceability check process is represented in figure 1. It consists of the following steps:

[2] OOD was developed by Taegyun Kim, at Pusan University of Foreign Studies, Pusan, Korea. It supports the construction of object diagrams defined in OMT.

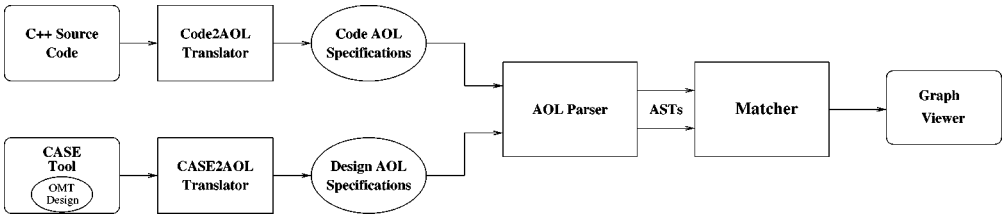


Figure 1. Design to code matching process.

1. AOL Representation Extraction: the AOL textual representation can be recovered from both the design and the code through respectively a `Code2AOL` and a `CASE2AOL` translator.
2. AOL Representation Parsing: an `AOL Parser` produces the AST (Abstract Syntax Tree) which subsequent phases rely on.
3. Match between design and code representations: a `Matcher` module implements the traceability check; it includes a function to compute the edit distance between attribute names, an implementation of the maximum matching algorithm and a maximum likelihood classifier.
4. Result Visualization: a `Pair Difference Coloring` module graphically shows the results of matching, highlighting similarities and differences between classes in the design and in the code.

The programmer is then in charge of the final step, in which the design is modified to solve all outlined differences from the code. This phase cannot be completely automated since the reasons for the major differences have to be fully understood by the designer, in order to perform a meaningful update of the design.

Additional information can be obtained by the designer, by querying the traceability links, and the associated measures, for the attributes of individual classes, by evaluating the traceability of the relations between classes, and finally by constructing the dictionary used for the names which compose the identifiers in the design and in the code.

### *3.1. AOL representation extraction*

AOL has been designed to capture OO concepts in a formalism independent of programming languages and tools. AOL is a general-purpose design description language, capable of expressing concepts available at the design stage of OO software development.

This language is based on the Unified Modeling Language (UML) [Rational Software Corporation 1997], a notation that is becoming the standard in object oriented design. UML is a visual description language with some limited textual specifications. Hence we designed from scratch many parts of the language, while remaining adherent to UML where textual specifications were available. At present, AOL covers only the

UML part related to class diagrams. Since, for class diagrams, the UML and OMT notations are almost identical, AOL is compatible with OMT designs.

The language resembles other OO design/interface specification languages such as IDL [Lamb 1987; OMG 1991] and ODL [Lea and Shank 1994]. However, AOL was thought of as a light C++ design representation language, thus it does not have the expressive power of ODL or IDL, but at the same time it allows a simple representation of all the class diagram concepts. Aggregations and associations are widely used in C++ programs and designs: AOL explicitly represents them while the former languages do not. As a result, a limited effort is required to develop a design or C++ code to AOL translator.

The AOL representation language ensures independence from any specific programming or proprietary design representation language. Future versions of our tool will incorporate the CDIF (CASE Data Interchange Format) standard, the common intermediate representation which has been recently developed by CASE tool vendors. More details on AOL can be found in [Antoniol *et al.* 1998; Fiutem and Antoniol 1998].

A `CASE2AOL Translator` module has been implemented for the StP/OMT [Interactive Development Environments 1996] tool, to obtain an AOL specification of the internal object models from the repository, while the `Code2AOL Translator` works on C++ code.

Extracting information about class relationships from code is far more difficult than from design: results might have some degree of imprecision. In fact, given two classes and a relation between them, there are intrinsic ambiguities, due to the choice left to programmers implementing the OO design, whether to consider such relation an association or an aggregation, unidirectional or bidirectional. Pointers, references, templates (e.g., `list<tree>`), arrays (e.g., `Heap a[MAX]`) can represent both associations and aggregations. A discussion on the semantics of the architectural information extracted from the code can be found in [Woods *et al.* 1999]. In the present work, an aggregation is recognized from the code if and only if an instance of an object is stored as a data member of another object or a template or object array data member is declared, even if a pointer could be used to create an object chunk. All the remaining cases, i.e., object pointers and references both as data members and formal parameters of methods, give origin to associations.

### *3.2. Test suite*

To assess the approach in an industrial environment an experiment of design-code compliance check was conducted on the design and the code of industrial software for telecommunications, provided by Sodalia SpA under the DEMOS 2 project. 29 components (about 308 KLOC, thousand Lines Of Code) were analyzed, for which both OO design models (stored as object models in the StP/OMT repository) and the corresponding code were available. All components were developed using the C++ language.

Table 1
Classes and attributes in the design and in the code for each component. LOC are given in the last column under the *Code* heading.

| | Design | | Code | | |
|---|---|---|---|---|---|
| Comp. | Classes | Attr. | Classes | Attr. | LOC |
| C1 | 38 | 280 | 53 | 1229 | 27398 |
| C2 | 17 | 374 | 16 | 416 | 19863 |
| C3 | 13 | 134 | 17 | 216 | 6190 |
| C4 | 113 | 438 | 103 | 1227 | 42781 |
| C5 | 7 | 82 | 26 | 188 | 15031 |
| C6 | 9 | 0 | 5 | 101 | 3428 |
| C7 | 29 | 139 | 21 | 496 | 21335 |
| C8 | 35 | 109 | 44 | 957 | 21796 |
| C9 | 29 | 329 | 39 | 644 | 11639 |
| C10 | 24 | 165 | 28 | 244 | 15319 |
| C11 | 18 | 260 | 18 | 249 | 14297 |
| C12 | 12 | 159 | 11 | 198 | 16847 |
| C13 | 1 | 19 | 2 | 50 | 2619 |
| C14 | 3 | 15 | 2 | 13 | 1081 |
| C15 | 12 | 0 | 7 | 174 | 11028 |
| C16 | 17 | 49 | 8 | 105 | 6116 |
| C17 | 6 | 29 | 6 | 64 | 438 |
| C18 | 12 | 346 | 15 | 335 | 20781 |
| C19 | 7 | 143 | 11 | 159 | 9913 |
| C20 | 9 | 131 | 8 | 233 | 9412 |
| C21 | 2 | 40 | 5 | 58 | 2561 |
| C22 | 9 | 25 | 4 | 26 | 2532 |
| C23 | 14 | 33 | 6 | 52 | 2514 |
| C24 | 16 | 219 | 16 | 229 | 9680 |
| C25 | 3 | 28 | 3 | 39 | 1422 |
| C26 | 9 | 57 | 6 | 55 | 3059 |
| C27 | 19 | 174 | 13 | 118 | 4922 |
| C28 | 7 | 54 | 6 | 51 | 2858 |
| C29 | 7 | 60 | 5 | 64 | 1447 |

Table 1 gives the number of classes and attributes in each analyzed component, as resulting from the design and from the code. LOC figures are shown in the last column under the *Code* heading. As one would expect, the number of classes and attributes extracted from the code is often substantially higher than in the design, thus indicating that the abstraction represented in the design typically ignores implementative details, like support classes and attributes which are introduced in the coding phase [Lorenz and Kidd 1994]. In a few cases the opposite is true (e.g., C2, C4, C7), and the higher number of classes in the design can be explained by the presence of context information inserted to help the programmer in understanding the whole operative setting of the classes under development.

Table 2
Deleted classes and average similarity measures for the design, as resulting from the maximum match algorithm.

| Comp. | Del. | Avg sim. | Comp. | Del. | Avg sim. |
|---|---|---|---|---|---|
| C1 | 0 | 0.829 | C16 | 9 | 0.909 |
| C2 | 1 | 0.944 | C17 | 0 | 0.856 |
| C3 | 0 | 0.992 | C18 | 0 | 0.947 |
| C4 | 10 | 0.636 | C19 | 0 | 0.983 |
| C5 | 0 | 0.986 | C20 | 1 | 0.905 |
| C6 | 4 | 0.963 | C21 | 0 | 0.988 |
| C7 | 8 | 0.748 | C22 | 5 | 0.815 |
| C8 | 0 | 0.698 | C23 | 8 | 0.871 |
| C9 | 0 | 0.884 | C24 | 0 | 0.808 |
| C10 | 0 | 0.968 | C25 | 0 | 0.996 |
| C11 | 0 | 1 | C26 | 3 | 0.877 |
| C12 | 1 | 0.907 | C27 | 6 | 0.948 |
| C13 | 0 | 0.983 | C28 | 1 | 0.874 |
| C14 | 1 | 0.628 | C29 | 2 | 0.994 |
| C15 | 5 | 0.890 | | | |

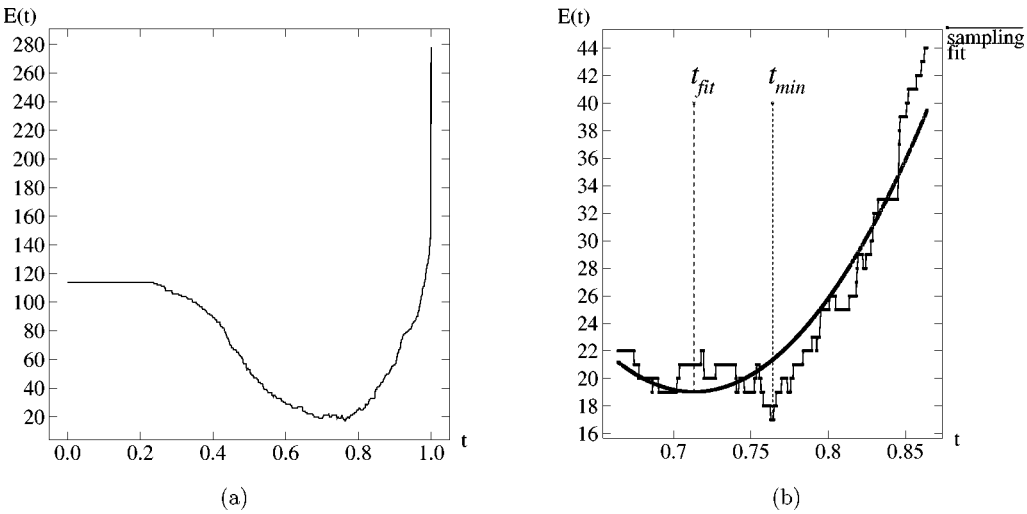


Figure 2. Finding the classification threshold. (a) The misclassification error $E(t)$, with component C1 excluded, is plotted as a function of the classification threshold, $t$. (b) Quadratic fitting of $E(t)$ in the neighborhood of $t_{\min}$.

### 3.3. *Average match figures*

Table 2 shows the results of the code traceability check, before applying the maximum likelihood classifier. The number of unmatched (deleted) classes in this table only accounts for the classes in the design for which the maximum match algorithm does not produce a traceability link. The application of the maximum likelihood

classifier may increase such a number and will be discussed in the following. The average similarity measure, i.e., the arithmetic average of the similarities in the recovered traceability links, is given for each component. This measure is also expected to increase after the application of the maximum likelihood classifier.

Average similarity is higher than 0.8, with the exception of components C4, C7, C8 and C14, while the maximum number of classes deleted from the design is 10. The low similarity (0.628) of component C14 was further investigated. The design of this component contains 3 classes. One of them is context information that is deleted. The remaining two classes have, respectively, a perfect match (similarity 1) and a very poor match (similarity 0.257) with code. A close inspection into the latter class reveals that the low similarity level is associated to a class to be considered actually unmatched, i.e., introduced to provide context information but having no counterpart in the code.

The maximum likelihood classifier was then applied to obtain a better identification of matched and unmatched classes. To evaluate its performance the cross validation technique was used. One component is taken out from the database, and a value, $t_{\min}$, for the classification threshold ($t$) is estimated on the remaining 28 components. The process is then iterated over all the components of the database, and at each iteration the accuracy of the classifier is assessed by counting the misclassification errors made on the excluded component.

As shown by the plot in figure 2(a), the large-scale behaviour of the error $E(t)$ is pretty smooth; yet, its "true" point of minimum (that is, abstracted from the sampling noise) can be localized only approximately. This may suggest that a more robust estimate of the classification threshold could be obtained by locally approximating $E(t)$ with a (low-degree) polynomial, and by setting the threshold to a value, $t_{\text{fit}}$, corresponding to the point of minimum of such a polynomial. In figure 2(b), a second degree fit of $E(t)$ is shown.

Results collected in table 3 show how the estimate of the classification threshold via polynomial fit does not improve the overall accuracy of the classifier; yet, it typically gives a more balanced occurrence of false positives and false negatives (see, for example, component C8), resulting in a higher robustness.

The average threshold computed by the maximum likelihood classifier using raw data (table 3, second column) is 0.74. This value is extremely stable with respect to the choice of the components used for its computation (the standard deviation is 0.03), and its value is not critical in the error minimization process (see figure 2). Therefore it can be used as a reference value for new components designed and developed in the considered industrial environment.

If such a value is used to classify the matched class pairs resulting from the application of the maximum match algorithm (see table 2), the number of classes deleted from the design and the average similarity measure become those in table 4.

The average similarity is substantially higher, with a minimum of 0.839, and above 0.9 with the exception of two components (C4 and C7). A perfect match is achieved for 3 components (C11, C14, C15), of which only one was already evident in table 2. This result was obtained by considering class pairs below the threshold as

Table 3
Columns 2–4 report results on raw data; columns 5–7 those obtained with the polynomial fit.

| | Raw data | | | Polynomial fit | | |
|---|---|---|---|---|---|---|
| Component | $t_{\min}$ | Precision | Recall | $t_{\text{fit}}$ | Precision | Recall |
| C1 | 0.764 | 1.0 | 0.926 | 0.738 | 0.954 | 0.971 |
| C2 | 0.764 | 1.0 | 1.0 | 0.715 | 0.95 | 0.976 |
| C3 | 0.764 | 1.0 | 1.0 | 0.706 | 0.95 | 0.976 |
| C4 | 0.764 | 0.912 | 0.756 | 0.74 | 0.97 | 0.992 |
| C5 | 0.765 | 1.0 | 1.0 | 0.717 | 0.951 | 0.977 |
| C6 | 0.763 | 1.0 | 1.0 | 0.714 | 0.951 | 0.977 |
| C7 | 0.697 | 0.857 | 0.923 | 0.717 | 0.957 | 0.979 |
| C8 | 0.695 | 0.588 | 1.0 | 0.701 | 0.973 | 0.983 |
| C9 | 0.765 | 1.0 | 1.0 | 0.714 | 0.947 | 0.975 |
| C10 | 0.763 | 1.0 | 1.0 | 0.694 | 0.949 | 0.982 |
| C11 | 0.687 | 1.0 | 1.0 | 0.723 | 0.955 | 0.972 |
| C12 | 0.686 | 1.0 | 1.0 | 0.713 | 0.951 | 0.976 |
| C13 | 0.764 | 1.0 | 1.0 | 0.714 | 0.952 | 0.977 |
| C14 | 0.764 | 1.0 | 1.0 | 0.728 | 0.958 | 0.97 |
| C15 | 0.764 | 1.0 | 1.0 | 0.707 | 0.951 | 0.977 |
| C16 | 0.764 | 1.0 | 1.0 | 0.721 | 0.957 | 0.973 |
| C17 | 0.695 | 1.0 | 1.0 | 0.705 | 0.951 | 0.977 |
| C18 | 0.747 | 0.917 | 1.0 | 0.717 | 0.953 | 0.976 |
| C19 | 0.764 | 1.0 | 1.0 | 0.688 | 0.948 | 0.983 |
| C20 | 0.763 | 1.0 | 1.0 | 0.71 | 0.951 | 0.977 |
| C21 | 0.694 | 1.0 | 1.0 | 0.726 | 0.958 | 0.974 |
| C22 | 0.764 | 1.0 | 1.0 | 0.698 | 0.952 | 0.983 |
| C23 | 0.764 | 1.0 | 1.0 | 0.696 | 0.952 | 0.983 |
| C24 | 0.763 | 1.0 | 1.0 | 0.7 | 0.951 | 0.983 |
| C25 | 0.694 | 1.0 | 1.0 | 0.711 | 0.952 | 0.977 |
| C26 | 0.763 | 1.0 | 1.0 | 0.724 | 0.957 | 0.973 |
| C27 | 0.764 | 1.0 | 1.0 | 0.732 | 0.956 | 0.969 |
| C28 | 0.763 | 1.0 | 1.0 | 0.733 | 0.957 | 0.97 |
| C29 | 0.764 | 1.0 | 1.0 | 0.709 | 0.951 | 0.977 |

unmatched. Consequently, the number of classes deleted from the design increases, but in several cases a unitary increase suffices to improve the design-code match (e.g., C2, C9, C10, C14).

A final revision of the traceability links is anyhow required from the designer, since the maximum likelihood classifier may in some infrequent cases be wrong, as shown by the values of precision and recall in table 3.

### *3.4. Design-code match of the relations*

Starting from the matching between the design and code classes obtained with the maximum match algorithm, the subset of the truly matched classes is considered. For each pair of matched classes, the relations of generalization, association and aggregation in the design have been searched for in the code. For each component and

Table 4
Deleted classes and average similarity measures for the design, as resulting from the maximum likelihood classifier.

| Comp. | Del. | Avg sim. | Comp. | Del. | Avg sim. |
|---|---|---|---|---|---|
| C1 | 11 | 0.956 | C16 | 10 | 0.982 |
| C2 | 2 | 0.984 | C17 | 1 | 0.966 |
| C3 | 0 | 0.992 | C18 | 0 | 0.947 |
| C4 | 74 | 0.839 | C19 | 0 | 0.983 |
| C5 | 0 | 0.986 | C20 | 2 | 0.996 |
| C6 | 4 | 0.963 | C21 | 0 | 0.988 |
| C7 | 15 | 0.856 | C22 | 6 | 0.990 |
| C8 | 18 | 0.909 | C23 | 9 | 0.988 |
| C9 | 1 | 0.900 | C24 | 5 | 0.984 |
| C10 | 1 | 0.995 | C25 | 0 | 0.996 |
| C11 | 0 | 1 | C26 | 4 | 0.999 |
| C12 | 3 | 0.996 | C27 | 7 | 0.999 |
| C13 | 0 | 0.983 | C28 | 2 | 0.999 |
| C14 | 2 | 1 | C29 | 2 | 0.994 |
| C15 | 6 | 1 | | | |

Table 5
Number of the generalization, association, aggregation relations of the design classes found between the corresponding code classes over the number of relations present in the design.

| Comp. | Gen. | Ass. | Agg. | Comp. | Gen. | Ass. | Agg. |
|---|---|---|---|---|---|---|---|
| C1 | 1/1 | 1/4 | 3/3 | C16 | 0/0 | 0/7 | 0/0 |
| C2 | 9/9 | 0/7 | 1/1 | C17 | 2/2 | 0/2 | 0/1 |
| C3 | 8/8 | 0/3 | 1/1 | C18 | 9/9 | 0/5 | 0/0 |
| C4 | 7/11 | 1/13 | 0/11 | C19 | 5/5 | 0/5 | 0/0 |
| C5 | 5/5 | 1/1 | 0/0 | C20 | 5/5 | 0/1 | 0/0 |
| C6 | 0/0 | 3/4 | 0/0 | C21 | 0/0 | 0/0 | 0/0 |
| C7 | 0/0 | 0/12 | 5/5 | C22 | 0/0 | 0/1 | 0/1 |
| C8 | 0/0 | 0/7 | 1/4 | C23 | 0/0 | 0/2 | 0/3 |
| C9 | 20/20 | 4/5 | 1/2 | C24 | 2/2 | 2/2 | 1/7 |
| C10 | 14/14 | 13/24 | 0/0 | C25 | 1/1 | 0/1 | 0/0 |
| C11 | 12/12 | 0/1 | 2/5 | C26 | 2/2 | 0/2 | 0/0 |
| C12 | 5/5 | 0/1 | 0/2 | C27 | 9/9 | 0/1 | 1/1 |
| C13 | 0/0 | 0/0 | 0/0 | C28 | 2/2 | 0/2 | 0/0 |
| C14 | 0/0 | 0/0 | 0/0 | C29 | 0/0 | 0/1 | 2/3 |
| C15 | 3/3 | 0/0 | 0/3 | | | | |

type of relation, the number of matching relations found in the corresponding code classes over the total number of relations between matched classes in the design are shown in table 5.

The traceability of the generalization relation is nearly total. Only in the C4 component there are 4 generalizations that do not appear in the code. Less traceable

are the association and aggregation relations (although the latter are better). The poor traceability of associations and aggregations has a twofold explanation. On one side the meaning of such relations, as intended by the designer, is loosely coupled with its implementation, being mainly associated to a vague concept of collaboration between design entities. On the other side, some relations are missing in the code due to the limitations of the reverse engineering tool, which is not always able to recognize them in complex constructs and data structures, and therefore does not retrieve some of the actually implemented relations. Generalization is the simplest relation to reverse engineer, and the ones missing in the code correspond to an incorrect representation of the underlying class hierarchy in the design. Association is the most difficult relation to recover, since it may be implemented by using *key-identifiers*, typically integers or strings, not directly related to the type of the associated objects. Aggregations are usually simpler to extract than associations because they are implemented through more standard programming constructs (e.g., arrays).

Component C4 is responsible for the graphical user interface, and consequently it mixes lots of automatically generated code, library classes and user code. Its traceability level is the lowest (0.839), with the maximum number of deleted classes (74), even after applying the maximum likelihood classifier. The traceability of its relations is again the lowest, with 4 generalizations and almost all associations and aggregations missing. The difficulties in separating relevant information from automatically generated code and library classes suggest that results on this component should be carefully interpreted. The true traceability level of this component would be expected to increase if the designers and developers of this component were involved to identify and compare the relevant portion of the design and the actually implemented code.

Summarizing, 121/125 (96.8%) generalizations, 25/114 (21.9%) associations and 18/53 (33.9%) aggregations can be automatically traced from the design to the code. If component C4 is excluded, of the 114 generalizations, 101 associations, and 42 aggregations present in the design, respectively 114 (100%), 24 (23.7%), and 18 (42.8%) can be found in the code.

### *3.5. Detailed analysis of an example component*

Average match figures discussed in the previous sections do not account for the finer grain information from which they have been computed. In fact, for each component traceability links between classes are available; moreover, for each pair of matched classes, attributes can be considered, and the overall similarity measure obtained for the classes can be split into matches between attributes from the design and from the code, each with an associated similarity measure. Programmers can use such a fine grain information to update designs making them consistent with the implementation. On the basis of the similarity values of the matched classes and attributes, they can decide to update names, and to add or remove classes and attributes.

To give a deeper insight into the process of evolving a design to improve its code traceability, component C16 was considered in more detail. Its average traceability

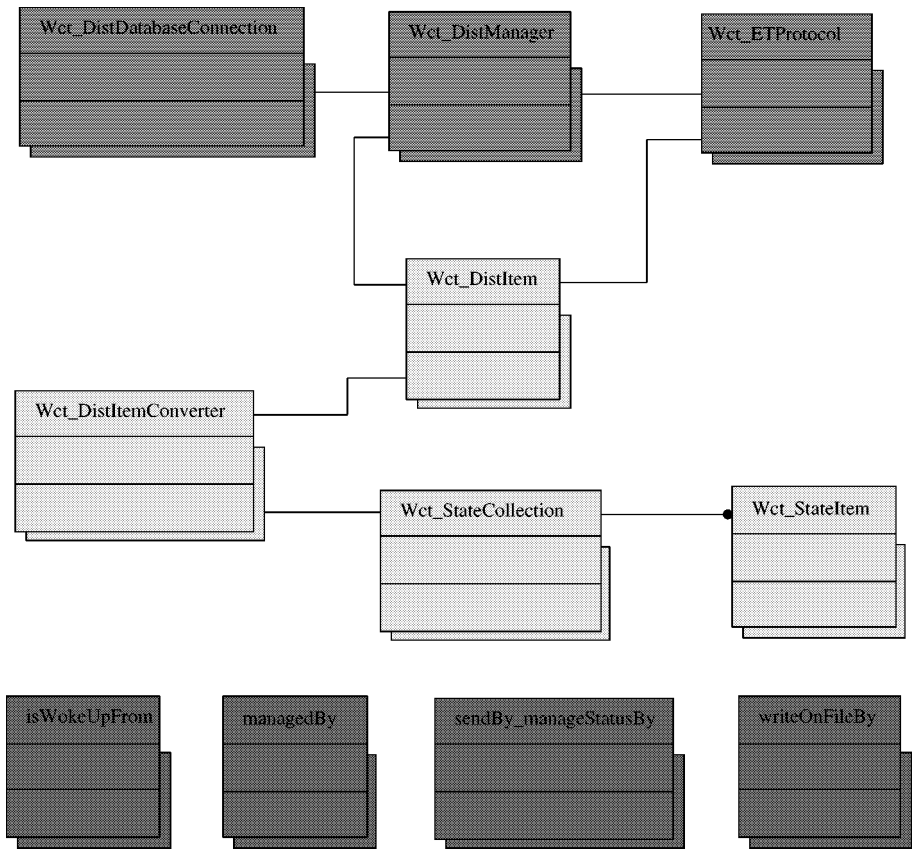


Figure 3. Pair difference diagram for component C16. Green classes are the three at the top (grey), green to yellow ones are in the middle (light grey), while some of the red classes are shown at the bottom (dark grey). Attributes have been omitted for clarity.

index is 0.982 with 10 unmatched classes, as resulting from the maximum likelihood classifier, which increases the number of deleted classes from 9 to 10 due to a very low match (0.397) between two classes. Manual inspection confirms that such additional deletion is correct. Figure 3 shows the graphical result of code traceability analysis for C16. Green boxes, used to represent a perfect match of classes in the design and in the code, are shown at the top (grey). Light-green to yellow is for the intermediate similarity levels, and accounts for the four classes in the middle (light grey, similarity levels 0.957, 0.985, 0.991 and 0.940). Red is used for the boxes at the bottom, associated to some of the unmatched classes (dark grey). One of them is an outcome of the maximum likelihood classification (similarity 0.397).

It can be noted that every non-red class has a prefix `Wct_` in its name. Therefore the hypothesis can be made that the red classes at the bottom are actually context information.

The match between the design and the code for the four light-green to yellow

Table 6
Attribute match for class Wct_DistItem from component C16. For each pair of attributes the associated similarity measure is shown.

| Design attribute | Code attribute | Sim. |
|---|---|---|
| Wct_DistItem | Wct_DistItem | 1 |
| ∼Wct_DistItem | ∼Wct_DistItem | 1 |
| deleteItemFromDirectory | deleteItemFromDirectory | 1 |
| extraction | extraction | 1 |
| init | init | 1 |
| loadItemStatus | loadItemStatus | 1 |
| setItemStatusToSENT | setItemStatus | 0.900 |
| deleteItemFromDatabase | loadItemNumberFromDatabase | 0.763 |

classes was then considered in more detail. Class Wct_DistItem has an average similarity measure equal to 0.957 with a same name class in the code. The matched attributes have been examined, as shown in table 6. In the last column the individual similarity measures are given for them.

For the first 6 attributes a perfect match with the code could be retrieved. The last but one attribute, setItemStatusToSENT, is matched with setItemStatus (similarity 0.900). The difference between the two names is in the suffix ToSENT specified in the design and omitted in the code. The operation implemented in the code is probably more general than the one in the design, and an additional parameter can be used to obtain the specialization represented in the design. It is likely that programmers recognized the opportunity of being more general with only a minor overhead, and therefore implemented a higher level function compatible with the needs of the given component, but more flexible and more reusable. The relatively low similarity level (0.763) of the last attribute is a hint suggesting a possible deletion of an operation when moving to coding. An examination of the attribute names seems to confirm such a hypothesis.

Evolving the design of class Wct_DistItem can therefore be obtained by replacing the name of attribute setItemStatusToSENT with setItemStatus and deleting attribute deleteItemFromDatabase. If such operations are performed, a perfect match between the design and the code is reached.

When the other three classes in the middle of figure 3 were considered in detail and the match between their attributes was examined, considerations similar to those for class Wct_DistItem could be made. In one case (class Wct_StateCollection) one of the non-perfect matches between attributes is due to a typing error.

### 3.6. *Design and code dictionaries*

After building design and code dictionaries for all components, the presence of words defined by the designer and not used by the programmer was checked. Only one occurrence of this case was found. Indeed, it corresponds to a word that, being misspelled in the design, was corrected when transiting to the code. However, the

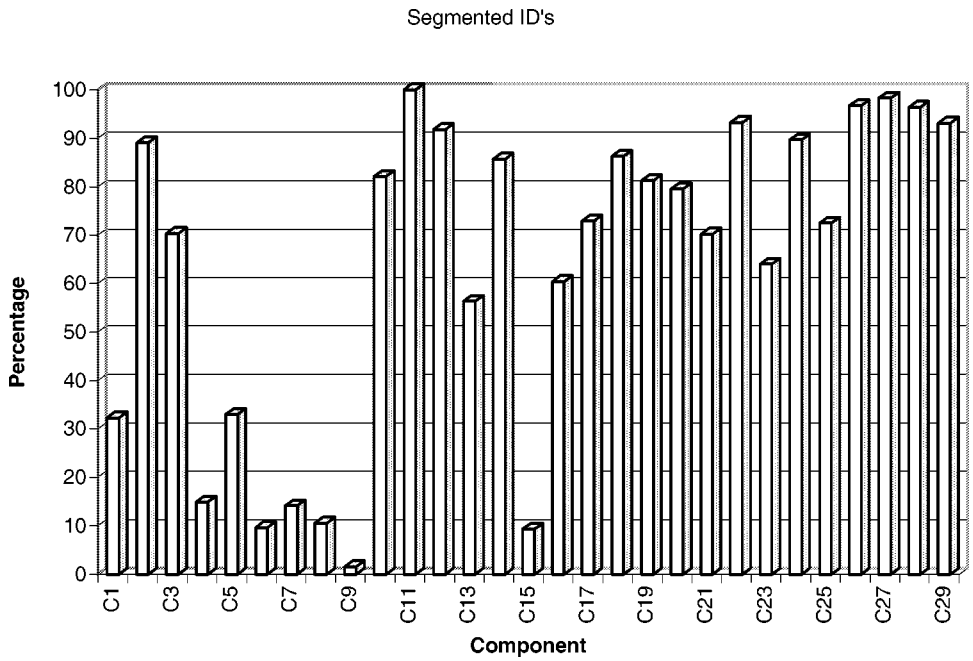


Figure 4. Percentage of code identifiers correctly segmented by using the design dictionary.

number of words added in the code is fairly high, with the noticeable exception of component C11, for which exactly the same dictionary was used in the design and the code. The average design dictionary size, over all the analyzed components, is 49.5 words (resulting from 134.1 identifiers), while the code dictionary size is 103.7 words (from 327.5 identifiers).

A second traceability indicator extracted through the lexical analysis of identifiers is the amount of code identifiers that can be segmented correctly by using the design dictionary (figure 4). The 100% segmentation rate obtained for component C11 does not come as a surprise. More interesting is the case of component C9: a moderate increment in the number of words (from 127 to 199) makes it impossible to segment the 98.5% of the code identifiers. In the typical cases, low segmentation rates are related to the insertion of several new words.

Our analysis seems to suggest that adoption of standard dictionaries, forcing the developers to choose from a constrained pool of terms, may significantly contribute to the enhancement of identifiers traceability. In this perspective, the second analysis performed on the test suite was directed to a better understanding of the nature of the dictionaries employed. We started by clustering words occurring in the component dictionaries into 9 distinct sets. Namely: English Forms (EF), as "*got*" or "*users*"; Special Strings (SS), as "<<"; Words Contractions (WC), as "*msg*" (message); Phrase Contractions (PC), as "*cbk*" (call back); Acronyms (A), as "*rpc*" (remote procedure call); Isolated Characters (IC); Misspelled and Non-English (MNE); Numbers (N); Others (O).

Table 7
Each column refers to one of the different types of words. The number of words in the cumulative dictionary (union of all component dictionaries) is given, while the sum over all the component dictionaries is between parentheses.

| | EF | SS | WC | PC | A | IC | MNE | N | O |
|---|---|---|---|---|---|---|---|---|---|
| Design | 613 (1813) | 9 | 113 (306) | 5 | 10 | 17 | 20 | 29 | 108 (193) |
| Code | 868 (2823) | 10 | 196 (540) | 8 | 17 | 23 | 34 | 30 | 108 (328) |

In table 7, the outcome of the clustering is reported as resulting from the cumulation of all the component dictionaries. Somewhat unexpectedly, the English Forms receive the lion's share, amounting to about five times the Words Contractions. Limited is the occurrence of Phrase Contractions and Acronyms. Definitely more significant, and possibly alarming, is the presence of many dozens of Others (words that, by visual inspection, could not be classified in any of the other categories), and of quite a few Isolated Characters (which, in some instance, may hint at *extreme* word contraction).

The high number of English Forms is an indicator of good dictionary traceability. In fact, the use of synonyms or word changes for English Forms is expected to be very limited. Words Contractions require on the contrary a closer manual inspection, since they are more likely to exhibit equivalent variations. They were signalled to the users for further investigation.

## 4. Related work

This paper extends previous works [Antoniol *et al.* 1999; Fiutem and Antoniol 1998] in that a different approach to compute the traceability links is adopted, now being based on the maximum match algorithm applied to similarity measures between attributes. In addition, the use of a maximum likelihood classifier makes no longer necessary the clean up phase. In fact, unmatched classes are determined as a combination of the outputs from the maximum match and the maximum likelihood classification algorithms, rather than being produced by a manual cleaning of design and code representations.

Among the works about model-implementation compliance checking that have been proposed [Luckham *et al.* 1987; Meyers *et al.* 1993; Murphy *et al.* 1995; Schwanke 1991; Sefika *et al.* 1996] in the literature, here we concentrate on those closest to our approach, i.e., those that explicitly address the problem of checking design against implementation and are applicable in the object oriented domain.

The work by Meyers et al. [1993] differs from ours both in the objective and in the implementation. The objective of CCEL is to check the compliance of a program against a set of design guidelines expressed as constraints that affect single or groups of classes, while our objective is to check the compliance of a design model against its implementation. Unlike CCEL, we have an explicit design model which states the existence of a set of specific entities with specific properties and relations among them and this must be verified in the implementation.

The work by Murphy *et al.* [1995] is much closer to ours. Software reflexion models can be applied in the OO domain: Murphy et al. refer to an experiment on an industrial subsystem where a reflexion model was computed to match a design expressed in the Booch notation against its C++ implementation.

Their process and ours are similar and many analogies can be drawn. We both use an extraction tool to derive abstract information from source code. The reflexion model tool is analogous to our design-code matcher, in that they both provide an output in terms of where the high-level model agrees or disagrees with the source code model. Where their and our approach mainly differ is in the use of the mapping between the two models. They use such mapping to trace the source code model entities onto the high-level model entities. But the nature and granularity of the two models is quite different: this is why such a mapping is needed. For example, they have modules in the high-level model and functions in the source code model: the mapping information is used to cluster the source code model entities in order to assign them to the high-level model entities. For this purpose they make use of regular expressions and exploit naming conventions of source code entities. In our case, the entities of the two models are exactly the same: classes and relations among them, and matching is based on similarity of properties, thus allowing a *partial* matching between entities in the two models. In fact, coding standards, naming conventions and programming style may alter design names to accommodate implementation details or shortcuts.

Pattern-Lint, the system developed by Sefika *et al.* [1996], differs from our work and Murphy's, which are based only on static analysis, in that it also integrates dynamic visualization, whose results are compared to the static-analysis ones. Although it is a very general and powerful framework, Pattern-Lint is able to check compliance of source code with respect to three types of design models: coding guidelines, architectural models such as design patterns or styles, and heuristic models such as coupling and cohesion. Pattern-Lint mainly differs from our work in that the tool is not provided with a high-level model representing all the system directly in terms of classes and relations to be compared with the corresponding information in the source code. Higher-level models or partial models, which represent pieces of a system, are provided to check compliance with specific parts of an implementation. Moreover, Pattern-Lint does not handle approximate matches like our system does using edit distance.

## 5. Conclusion

Design documents are often inconsistent with source code implementation. Maintaining consistency is a costly and tedious activity and reducing time to market is often vital to face competition. However, being able to trace design into code and to evolve design is fundamental in both development and maintenance phases of the software life cycle.

Automatic tools to support design-code compliance check, showing potential discrepancies and lack of traceability between the two artifacts are thus helpful to

drive design evolution. Reports about design-code compliance, and graphical plots such as the pair-difference diagrams, can be used for this purpose.

Industrial software, and especially that developed with OO technology, is often built using a component-based strategy or based on COTS and libraries. A design-code compliance check tool must take into account the "physiological" inconsistencies between design and code represented by reuse and COTS. To properly handle such cases, a maximum likelihood classifier was applied after the maximum match computation, to obtain an accurate set of classes to be considered unmatched.

The concept of *similarity* between entities in the design and in the code, relaxing the exact name matching and introducing an edit distance, was the key to find the best match even in cases in which modifications were introduced in the names of the attributes.

The design-code compliance check has been applied to an industrial system and it has allowed to obtain an average traceability of 0.971, with an average of 6.37 unmatched classes in the design. Before applying the maximum likelihood classifier the average traceability was 0.890 and the number of deleted classes 2.24.

The traceability of the relations between matched classes gives an overall value of 96.8% generalizations specified in the design and implemented in the code, 33.9% aggregations and 21.9% associations. The lower traceability of aggregations and associations is in part due to the intrinsic limitations of the reverse engineering tool which extracts them from C++ code.

In a case study the result of the match was considered in a finer detail. For two matching classes with similarity measure 0.982 the individual differences between attributes as specified in the design and as implemented in the code were analyzed in detail, and provided information useful to update the design. The evolved design for this component could reach a perfect match level (similarity 1) with the code.

Finally the dictionaries constructed for the design and the code revealed that the words used by the designer to build the identifiers make up also the dictionary used in the code, with some extensions, thus confirming the important role of names in the cases in which traceability is only implicitly supported.

## Acknowledgements

The authors would like to thank Sodalia SpA for the support provided to this research. They are also grateful to Giuseppe La Commare and Roberto Fiutem for the seminal ideas and discussions which highly contributed to the development of this activity.

## References

Antoniol, G., R. Fiutem, and L. Cristoforetti (1998), "Using Metrics to Identify Design Patterns in Object-Oriented Software," *Proc. of the Fifth International Symposium on Software Metrics – METRICS'98*, pp. 23–34.

Antoniol, G., A. Potrich, P. Tonella, and R. Fiutem (1999), "Evolving Object Oriented Design to Improve Code Traceability," In *Proc. of the International Workshop on Program Comprehension*, Pittsburgh, PA, pp. 151–160.

Ball, T. and S.G. Eick (1996), "Software Visualization in the Large," *IEEE Computer 29*, 4, 33–43.

Baxter, I.D. and C.W. Pidgeon (1997), "Software Change Through Design Maintenance," In *Proc. of the International Conference on Software Maintenance*, Bari, Italy, pp. 250–259.

Bunge, M. (1977), *Treatise on Basic Philosophy*, Vol. 3: *Ontology I: The Furniture of the World*, Reidel, Boston, MA.

Bunge, M. (1979), *Treatise on Basic Philosophy*, Vol. 4: *Ontology II: A World of Systems*, Reidel, Boston, MA.

Chidamber, S.R. and C.F. Kemerer (1994), "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering 20*, 6, 476–493.

Cormen, T.H., C.E. Leiserson, and R.L. Rivest (1990), *Introductions to Algorithms*, MIT Press.

Duda, R.O. and P.E. Hart (1973), *Pattern Classification and Scene Analysis*, Wiley, New York.

Fiutem, R. and G. Antoniol (1998), "Identifying Design-Code Inconsistencies in Object-Oriented Software: A Case Study," In *Proceedings of the International Conference on Software Maintenance*, Bethesda, Maryland, pp. 94–102.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, Reading, MA.

Garlan, D. and M. Shaw (1996), *Software Architecture: Perspectives on an Emerging Discipline*, Vol. 1, Prentice-Hall, Englewood Cliffs, NJ.

Holt, R. and J.Y. Pak (1996), "GASE: Visualizing Software Evolution-in-the-Large," In *Proceedings of the Working Conference on Reverse Engineering*, Monterey, pp. 163–166.

Interactive Development Environments (1996), *STP Manuals*.

Lamb, D.A. (1987), "IDL: Sharing Intermediate Representations," *ACM Transactions on Programming Languages and Systems 9*, 3, 297–318.

Lea, D. and C.K. Shank (1994), "ODL: Language Report," Technical report, NY CASE Center.

Lindvall, M. and K. Sandahl (1996), "Practical Implications of Traceability," *Software: Practice and Experience 26*, 10, 1161–1180.

Lorenz, M. and J. Kidd (1994), *Object-Oriented Software Metrics*, Prentice-Hall, Englewood Cliffs, NJ.

Luckham, D., F. von Henke, B. Krieg-Bruckner, and O. Owe (1987), "Anna, A Language for Annotating Ada Programs: Reference Manual," In Lecture Notes on Computer Science, Vol. 260, Springer-Verlag, Berlin, Germany.

Meyers, S., C.K. Duby, and S.P. Reiss (1993), "Constraining the Structure and Style of Object-Oriented Programs," Technical Report CS-93-12, Brown University.

Murphy, G.C., D. Notkin, and K. Sullivan (1995), "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models," In *Proceedings of the Third ACM Symposium on the Foundations of Software Engineering*, pp. 18–28.

OMG (1991), "The Common Object Request Broker: Architecture and Specification," Technical Report 91.12.1, Object Management Group.

Rational Software Corporation (1997), *Unified Modeling Language, Version 1.0*.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991), *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ.

Schwanke, R. (1991), "An Intelligent Tool for Reengineering Software Modularity," In *Proceedings of the International Conference on Software Engineering*, pp. 83–92.

Sefika, M., A. Sane, and R.H. Campbell (1996), "Monitoring Compliance of a Software System with its High-Level Design Models," In *Proceedings of the International Conference on Software Engineering*, pp. 387–396.

Woods, S., S.J. Carriere, and R. Kazman (1999), "A Semantic Foundation for Architectural Reengineering and Interchange," In *Proc. of the International Conference on Software Maintenance*, Oxford, England.