# Assessing the Relevance of Identifier Names in a Legacy Software System*

Nicolas Anquetil Timothy Lethbridge
School of Information Technology and Engineering
150 Louis Pasteur, University of Ottawa
Ottawa, Canada, K1N 6N5
(1) (613) 562-5800 x6688
{anquetil,tcl}@site.uottawa.ca

## Abstract

Reverse engineering is a difficult task even for humans. When trying to provide tools to assist in this task, one should try to take advantage of all possible sources of information.

Informal sources, like naming conventions, are more abstract than the code, thus helping to bridge the gap between code and design. On the other hand, there is no certainty that they actually represent the current state of the system.

Some researchers have used these informal sources of information, relying on the fact that types (or variables, or functions) have the same name to assume they represent the same thing.

However none of these researchers actually tried to establish to what extent this assumption was verified.

This paper aims at providing a framework to study this point. We will define what it means to have a "reliable naming convention", how this can be tested and under what conditions. Examples from the particular legacy software system we are studying as well as from the literature are presented.

## 1 Introduction

Maintaining legacy software systems is a problem which many companies face. To help software engineers in this task, researchers are trying to provide tools to help recover the design structure of the software system. In general, existing research focuses on the source code as the place from which to extract design concepts.

Some researchers however considered more informal sources of information of a higher level of abstraction [3, 5, 7, 13]. The difficulty with these is that they may be the remnants of an old design no longer representing the actual state of the system. Therefore one should be careful when basing assumption on it.

In [2] we showed that file names can provide a reliable source of information to extract subsystems. But is naming convention still reliable when it comes to identifiers?

The community appears divided on this issue. In [17], Sneed states that "in many legacy systems, procedures and data are named arbitrarily". Other researchers have used heuristics based on naming convention (eg. [4, 6, 15]) but without assessing formally their efficiency.

This paper aims at providing a framework to study this point. We will define what it means to have a "reliable naming convention", how this can be tested and under what conditions. Examples from the particular legacy software system we are studying.

## 2 Naming Convention and Reverse Engineering

Reverse engineering is a difficult task even for humans. When trying to provide tools to assist in this task, one should take advantage of all possible sources of information.

Informal source, like naming conventions, are more abstract than the code, thus helping to bridge the gap between code and design. On the other hand, there is no certainty that they actually represent the state of the system.

In [3] we used file naming convention to extract subsystems in the legacy system we work with. Our results show that file names are a useful source of information in this case [1, 2].

In the case of identifier names, opinions are contradictory: Sneed [17] reports that "programmers often choose to name procedures after their girlfriends or favorite sportsmen", whereas other researchers have used heuristics based on the assumption that identifiers names were somehow significant [4, 6, 15].

There are different kinds of identifiers: variables, user defined types, functions, etc. Because they are used to represent very different things (algorithms, abstract data types, ...), each kind would probably require a different treatment. In this paper we will focus on (user defined) structured types and their fields.

Structured types are of particular interest in reverse engineering because they represent abstract data types which are good candidates to form classes (see for example [6, 9, 12, 15, 20]). One difficulty with structured types is to detect when two of them, possibly slightly different, implement the same abstract data type:

- Sneed again, states that "data attributes of the same structure may have different names from one program to another".

- Newcomb [15] or Cimitile et al. [6] are comparing the definitions of the structured types to find out the *synonyms*, i.e. structured types with different names but implementing the same abstract data type.

Relying on the structured type definitions, Newcomb and Cimitile assume that if the structured type names are not significant, the field names are. But they did not formally assess to what extent either assumption is true.

This paper aims at providing a framework to study this point. We will now propose a formal definition of "reliable naming convention".

### 2.1 Reliable Naming Convention

It should first be made clear that this work is somehow based on an important assumption: By trying to test how relevant naming conventions are (with regard to the design) we are supposing that the software engineers are trying to give significant names (although they may failed in this attempt).

As already mentioned, in Sneed's experience [17], this initial attempt was not made. He found legacy software where procedures were named after peoples (sportsmen or software engineers' girlfriends). One can only hope this kind of behavior is not the rule.

For us, we will assume that as long as identifiers are not people names, they do represent the concepts implemented. In our legacy system, names seem to have been chosen to help understanding what they represent.

The expression "naming convention" can be misleading because it suggests a preliminary agreement of the software engineers on how to name things. This would not be a realistic hope and it is not what we are aiming at.

Ideally, we would say that the *naming convention* is reliable if there is an equivalence between the name of the software artifacts and the concepts they implement. These concepts will be though of as domain or design concepts.

For example, for structured types, this would mean there is an equivalence between the name of a structured type and the abstract data type it implements.

The problem of comparing concepts (such as design concept) will be addressed in section §3. Even without considering this aspect, the above definition is very strict. In reality, we will have to consider independently each side of the equivalence:

- Two software artifacts with the same name should implement the same concept.

- A concept should have the same name for each of its different implementations. (Corollary: Two software artifacts with different names should implement different concepts.)

The second implication is a well known, unresolved, problem of "forward" engineering: "studies of how people name things have shown that the probability of having two people apply the same name to an object is between 7% and 18%, depending on the object" [8].

We cannot hope the software systems we will study solved it, therefore when talking about the reliability of naming convention, we will primarily focus on the first equivalence.

Before considering the difficult problem of comparing the concepts implemented, we will briefly describe the example software system we used for our experiments.

### 2.2 The Data

We are working on a real world telecommunication legacy software system. This system is over 15 years old and about 2 MLOCs of Pascal code. The system contains over 7000 structured type definitions (*records* in Pascal).

A record can be defined as a type and then be used to declare variables. Or it can be defined "on the fly" when declaring a variable. We call the second *anonymous* record definition, because it does not have a name of its own. We did not considered this second type of definition because obviously we cannot assert the reliability of such record names.

Also, records can be defined globally (by a program), or locally (by a function). We will only consider global record definitions.

There are 2666 global, non anonymous, record definitions, out of which, 97 records have a non-unique name. This is relatively small subset of all the record definitions. If we consider the global and local, non anonymous, record definitions, there are over 6000 records, with 542 common names covering 1709 records.

We will mainly be comparing the records pairwise. There are 44 common names, which yield 77 pairs of synonymous records:

- 40 names common to 2 records (40 synonymous record pairs),
- 3 names common to 3 records (9 synonymous record pairs), and
- 1 name common to 8 records (28 synonymous record pairs).

## 3 "Conceptual" Similarity Metrics

In establishing whether a naming convention is reliable or not, a very important point will be how we compare the concepts implemented.

Ideally we would have an oracle (human) to tell us whether two software artifacts with the same name implement the same concept.

This solution is not tractable given the possible size of the data (thousands of software artifacts). We will therefore rely on the source code to try to establish the similarity of two implemented concepts.

We suppose here that any change in the design will be reflected by a corresponding change in the implementation. This is the act of faith on which is based all the work in the reverse engineering community.

### 3.1 Comparison of Definitions

Intuitively it seems that the definition of a software artifact is the most appropriate handle (if not the sole one) to get a grip on the concept implemented. This is how Newcomb [15] and Cimitile et al. [6] detect synonymous records. This is also how Mayrand et al. [14] propose to detect clone functions.

Although we will not explore this path, we believe, one could also consider the uses of the software artifacts. The idea comes from the way cohesion and coupling of subsystems — which is basically measuring how close files in the subsystem are one from the other— are computed: One compares the types or global variables that are used in these files [11, 16].

Experience with these metrics suggests the results would be very similar. The two approaches should be very close in their implementation also as they would both come down to compare other names, of variables using the records in one case, of the records' fields in the other case.

We will only consider here the record definitions. The theoretical difficulty lies in the fact that one assumes the fields' names and types are reliable whereas the record names are not.

To justify this somehow incoherent assumption, Newcomb proposes a subjective argument

| Synonymous records | | Field Types | | | | total |
|---|---|---|---|---|---|---|
| | | = | | ≠ | | |
| Field | = | 73 | (94.8%) | 4 | (5.2%) | 77 |
| Names | ≠ | 52 | (11%) | 421 | (89%) | 473 |

Table 1: Paired comparison of fields' names and fields' types for synonymous records. The results are given in number of record pairs.

(accumulation of imperfect proofs): "For complex records consisting of 5-10 or more fields, the likelihood of false positives[1] is relatively small. For smaller records the probability of false positive is fairly large."

There are a number of problem with this argument:

- Although common sense suggests it is true, it would have to be formally established.
- It implies to fix a threshold (somewhere between 5 and 10 for Newcomb) which presumably would depend on the software system.
- It does not say much for records that are below the treshold.

In the system we are studying, the average size of the structured types we considered is 3.6 fields, which is well below Newcomb's threshold.

It would be easier if we knew "how much" reliable the field names are.

As for the records, we may try to assess the field names reliability by comparing their definition (i.e. the fields' types). Fortunately, the "recursive testing" may stop if we deal with typed programming languages. For these languages, the type of a variable (or a field in our case), is usually meaningful[2].

One must however make sure that:

- The type of the field is not not one of the basic types of the language (integer, boolean, characters, ...)
- The type of the field is not an anonymous record.

[1] i.e. Structured types assumed to represent the same abstract data type whereas it is not the case.

[2] There are differences among the typed languages. In C for example, the typing is looser than in Pascal.

We exclude the basic types of the language because they are too general and meaningless: We can make very few assumptions from the fact that two variables are integers.

We exclude anonymous records because to assert if they are equal, one would have to compare their fields and therefore assume the fields name are reliable which is what we are trying to establish.

For all the pairs of synonymous records in our system, we compared their fields' names and fields' types. The results are given in table 1. These numbers show that inside the synonymous records subset, the field names are reliable: A high proportion (94.8%) of synonymous fields have the same type (first implication of section §2.1, same name ⇒ same implemented concept), and the proportion of non synonymous fields with different types (89%) is also good (second implication of section §2.1, different names ⇒ different implemented concepts).

However, the set of synonymous records is a relatively small subset of all the records. To be of interest, the experiment should consider all the records.

These new results (table 2) are not as clear. At first it would seem that the first implication is not respected whereas the second (that we judged less likely to be true) is.

In fact, the overwhelming number of field pairs with different names or different types invalidate all the results. A possible solution would be to split down the system into subsystems and consider each subsystem independently. We will come back on this in section §4.

## 3.2 Field Based Similarity

We have established that inside the subset of records that have non unique names, the field

| All records | | Field Types | | | | total |
|---|---|---|---|---|---|---|
| | | = | | $\neq$ | | |
| Field | = | 7709 | (33.7%) | 15174 | (66.3%) | 22883 |
| Names | $\neq$ | 158828 | (0.2%) | 66652062 | (99.8%) | 66810890 |

Table 2: Paired comparison of fields' names and fields' types for all records. The results are given in number of record pairs.

naming convention is reliable. We will see how we can use such a result to formally compare records.

To compare the record definitions we will use similarity metrics from [10]. These metrics compare two lists of "attributes" (in our case, field names with their types). Given the two lists $l_1$ and $l_2$, the metrics are based on the following values $a = \|l_1 \cap l_2\|$, $b = \|l_1 \setminus l_2\|$ and $c = \|l_2 \setminus l_1\|$:

**Jaccard:** $\mathit{similarity}(l_1, l_2) = \dfrac{a}{a+b+c}$

**Sørensen-Dice:** $\mathit{similarity}(l_1, l_2) = \dfrac{2a}{2a+b+c}$

**Ochiai:** $\mathit{similarity}(l_1, l_2) = \dfrac{a}{\sqrt{(a+b)(a+c)}}$

When the need will appear to make a difference between the metrics themselves and the ways they are used to compare the record definitions, we will refer to these three as *similarity metrics* and to the way we use them as *conceptual similarity metrics* ("conceptual" because we use them to compare the "concepts" or abstract data types implemented by the records).

For the conceptual similarity metrics, the record definitions will be represented by lists of field names and types (together). Because we compare each field and its type as a whole, we will say these conceptual similarity metrics are *field based*. We will see in the next section conceptual similarity metrics based on words.

Table 3 (upper part) gives the distribution for the 77 pairs of synonymous records for the three metrics. The two special values zero (records completely different) and one (records completely equal) have been singled out to allow a better comparison of the results with following experiments.

First a good news: one can see that the results are very similar for the three metrics.

The bottom part gives the results when we do not take into account the type of the fields in the list describing each record. As expected, this makes little difference.

On the other hand, it would seem that the record names are not reliable. About half of the pairs have a low conceptual similarity ($<$ 0.4). This is in fact a flaw in our field based conceptual similarity metric.

### 3.3 Word Based Similarity

Some record pairs have fields with very close, yet different, names. As a result, the similarity metric consider such records completely different one from the other (null similarity).

The eight records sharing one single name ("general_software_log_data_type") provide a very good example of this. They have the following fields:

- {cpreport_table_id: language_files, cpreport_string_1: integer} (1 record),
- {hm_table_id: language_files, hm_string_1: integer} (2 records),
- {msgctrp_table_id: language_files, msgctrp_string_1: integer} (1 record),
- {smdr_table_id: language_files, smdr_string_1: integer} (1 record),
- {trfmtce_table_id: language_files, trfmtce_string_1: integer} (2 records),
- {trfreport_table_id: language_files, trfreport_string_1: integer} (1 record).

Because of the different prefixes in the field names, the records have a null similarity between them, whereas in fact, these small differences themselves reinforce the impression of

| | Similarity intervals | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | $]0, 0.2[$ | $[0.2, 0.4[$ | $[0.4, 0.6[$ | $[0.6, 0.8[$ | $[0.8, 1[$ | 1 |
| Jaccard | 32 | 2 | 1 | 2 | 3 | 5 | 32 |
| Sørensen-Dice | 32 | 0 | 3 | 0 | 4 | 6 | 32 |
| Ochiai | 32 | 0 | 3 | 0 | 4 | 6 | 32 |
| Jaccard | 32 | 1 | 1 | 2 | 2 | 5 | 34 |
| Sørensen-Dice | 32 | 0 | 2 | 0 | 3 | 6 | 34 |
| Ochiai | 32 | 0 | 2 | 0 | 3 | 6 | 34 |

Table 3: Conceptual similarity of the 77 synonymous record pairs. The results are given in number of pairs in each similarity interval. The top part was measured with fields' names and types, the bottom part with fields' names only.

intentional similarity. The records are not just simple copy/pastes to avoid rewriting a few lines of code. They were purposedly modified to mark minor differences while still pertaining to the same general abstract data type.

These eight records act as implementations of subtypes of a general abstract data type that would have the following two fields {table_id: language_files, string_1: integer}.

This single problem accounts for 26 pairs out of the 32 with null conceptual similarity (24 pairs for the name "general_software_log_data_type" and two other pairs from two other names).

There are possibly two ways to overcome this problem:

- consider only the field types and not the field names,
- allow imperfect match between field names.

We rejected the first solution on fear that we do not have enough data left to compare the records. Because we said we don't want to use the language basic types, we would have to eliminate the integer type in these fields and therefore compare the records on only one type ("language_files") which seems little. Presumably, some records would have nothing left to compare them (if all their fields have basic types).

We will rather consider the second solution and propose a word based conceptual similarity metric that would take into account the similitude between the fields' names.

To do this, we will simply decompose the field names into the list of their constituent words ("cpreport_table_id" decomposes in "cpreport", "table" and "id"), hence the name *word based* conceptual similarity metrics.

Decomposing the record names is not a difficult task with the system we are studying. As a rule, the identifiers include "word markers" (the underscore sign) on which to break them.

In addition to this, some identifiers include numbers like "pid0" or "hm_string_1". If these numbers or not isolated by "word markers" (in "hm_string_1", 1 is isolated; in "pid0", 0 is not), we consider them version number and decompose the word as follow: "pid0" gives "pid" and "pid0".

See [3] for a discussion on how to decompose names which do not contain "word markers"

There are different ways to compute the word based conceptual similarity metric. We could come up with a "weighted" version of the metrics, where $\|\{cpreport\_table\_id, cpreport\_string\_1\} \cap \{hm\_table\_id, hm\_string\_1\}\|$ would not be zero, but would consider that the elements of the lists have some similitude.

This solution would require many difficult decisions, as: what to do when some names from $l_1$ all imperfectly match some names from $l_2$? Deciding which names of $l_1$ will be matched with which other name of $l_2$ would be a difficult task that we did not want to undertake.

We propose an easier solution, the list representing each record definitions will be made of all the "words" composing the field names of the record. Thus, computing the word based

| | Similarity intervals | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | ]0, 0.2[ | [0.2, 0.4[ | [0.4, 0.6[ | [0.6, 0.8[ | [0.8, 1[ | 1 |
| Jaccard | 4 | 3 | 0 | 27 | 2 | 7 | 34 |
| Sørensen-Dice | 4 | 0 | 3 | 0 | 28 | 8 | 34 |
| Ochiai | 4 | 0 | 3 | 0 | 28 | 8 | 34 |
| Jaccard | 3 | 1 | 3 | 1 | 29 | 8 | 32 |
| Sørensen-Dice | 3 | 0 | 3 | 1 | 29 | 9 | 32 |
| Ochiai | 3 | 0 | 3 | 1 | 29 | 9 | 32 |
| Jaccard | 2 | 2 | 3 | 0 | 29 | 9 | 32 |
| Sørensen-Dice | 2 | 1 | 3 | 1 | 27 | 11 | 32 |
| Ochiai | 2 | 1 | 3 | 1 | 27 | 11 | 32 |

Table 4: Word based conceptual similarity of synonymous records. The top part was measured with decomposed field names only. The middle part was measured with decomposed field names and non decomposed field types. The bottom part was measured with decomposed field names and decomposed field types.

conceptual similarity between these two lists:

- {cpreport_table_id, cpreport_string_1}
- {hm_table_id, hm_string_1}

will consist in applying the similarity metric on the two following lists of "words":

- {cpreport, table, id, cpreport, string, 1}
- {hm, table, id, hm, string, 1}

This solution may appear a bit too simplistic at first because we loose the field to field comparison and just compare all the words. This means for example that the two lists {aa_11, bb_22} and {aa_22, bb_11} would get a "perfect" similarity (equal to one) whereas they are not equal.

To reduce the risk of such problem, we suggest to drop the fields' types for this experiment. Of course, one can do this only if the field names have proved to be reliable. If they are not, one could put the field types in the list as independent word themselves. The basic types of the language would be discarded. A priori, it does not seem fitted to also decompose the type names, but this could be subject to discussion.

Table 4 gives the results of the new experiment for our system. The top part was obtained with the lists characterizing each record containing only the decomposed field names. The middle part was obtained with the lists containing the decomposed field names and the non decomposed field types. Finally the bottom part was obtained with the lists containing the decomposed field names and the decomposed field types.

In this case, the three alternatives make little difference.

As expected, the results are better. The flaw we detected has been corrected and only 10% of the pairs have a conceptual similarity inferior to 0.6 . Two pairs are definitely different (null similarity). These are utility records. The number of perfectly equal pairs did not change (32 if we include the field types, 34 if we do not include them). This may indicate that the problem of "cross similarity" (between list like aa_11, bb_22 and aa_22, bb_11) we feared did not occur.

# 4 Naming Convention in Legacy Software

Because we are dealing with legacy software, there is another problem one should consider, that of *reliability over space*. Because legacy software are very large, different parts of the entire system (presumably subsystems) could have different naming conventions.

Consider the following example: In the system we study, there are 16 records with an attribute “data”[3]. Out of these 16 fields, only four have the same type (two pairs of “data” fields with the same type). These two similarly typed pairs occur in records that have either the same name or very similar names:

- two records “ss7msgdmp_trace_buf_item” have a “data” field of type “message”,

- records “registers” and “lld_registers” have a “data” field of type “data_register”.

The 12 other “data” fields (with dissimilar, non basic, types) occur in records that have dissimilar names.

We believe this is the same kind of localized naming conventions that caused the field names to appear non relevant over all the records (table 2, page 5) whereas they appear relevant over the subset of synonymous records (table 1, page 4).

These localized naming convention would make their testing all the more difficult because it imposes to know the partitioning of the system over which the different naming convention are based.

Presumably, the naming convention would be localized in subsystem, but there is no certainty about it. It could also depend on the software engineers who would be working on more than one subsystem.

# 5 Conclusion and Future Work

Being able to rely on the names of software artifacts to detect different implementations of the same concept would be very useful. It would for example allow to extract the “names” of the design concepts thus allowing the program comprehension tools to use the same language the software engineers do. It could also decrease the amount of data to deal with (considering records’ names instead of all the records’ fields).

Some researchers tried to rely on names in their reverse engineering effort. But they did not formally assess to what extent naming conventions are reliable in the systems they study.

In this paper we presented a framework to do so. We proposed a definition for “reliable naming convention” and proposed some experiments to evaluate it.

The possibly difficult issue of “consistency over space” (or “localized naming convention”) was raised.

We tried to illustrate our discussions with examples and experiments from the particular software system we study. But these were limited in size and do not allow to draw any significant conclusion for this system. An extension of this work would be to conduct real experiments over the system.

[3] In fact, there are 25 “data” fields, but we do not consider nine of them which use one of the language’s basic types.

# Thanks

The authors would like to thank the software engineers at Mitel who participated in this research by providing us with data to study and by discussing ideas with us. We are also indebted to all the members of the KBRE research group for fruitful discussions we have had with them.

# About the authors

Nicolas Anquetil completed is Ph.D. at the Université de Montréal in 1996. He is now working as a research associate and part time professor in the School of Information Technology and Engineering (SITE) at the University of Ottawa.

Timothy C. Lethbridge is an Assistant Professor in the School of Information Technology and Engineering (SITE) at the University of Ottawa. He leads the KBRE group, which is one of the projects sponsored by the Consortium for Software Engineering Research.

The authors can me reached by email at `{anquetil,tcl}@site.uottawa.ca`. The URL for the project is `http://www.site.uottawa.ca/~tcl/kbre`.

## References


[1] Nicolas Anquetil and Timothy Lethbridge. File Clustering Using Naming Conventions for Legacy Systems. In J. Howard Johnson, editor, *CASCON'97*, pages 184–95. IBM Centre for Advanced Studies, nov 1997.

[2] Nicolas Anquetil and Timothy Lethbridge. Design quality of subsystems extracted from file names. Technical Report TR-98-06, University of Ottawa, Computer Science Departement, MacDonald Hall, 150 Louis Pasteur, room 329, Ottawa, Canada, K1N 6N5, jul. 1998.

[3] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *International Conference on Software Engineering, ICSE'98*, pages 84–93. IEEE, IEEE Comp. Soc. Press, apr. 1998.

[4] Elizabeth Burd, Malcom Munro, and Clazien Wezeman. Extracting reusable modules from legacy code: Considering the issues of module granularity. In *Working Conference on Reverse Engineering*, pages 189–196. IEEE, IEEE Comp. Soc. Press, nov 1996.

[5] G. Butler, P. Grogono, R. Shinghal, and I. Tjandra. Retrieving information from data flow diagrams. In *Working Conference on Reverse Engineering*, pages 22–29. IEEE, IEEE Comp. Soc. Press, jul. 1995.

[6] A. Cimitile, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino. Identifying objects in legacy systems. In *5th International Workshop on Program Comprehension, IWPC'97*, pages 138–47. IEEE, IEEE Comp. Soc. Press, 1997.

[7] Julio Cesar Sampaio do Prado Leite and Paulo Monteiro Cerqueira. Recovering business rules from structured analysis specifications. In *Working Conference on Reverse Engineering*, pages 13–21. IEEE, IEEE Comp. Soc. Press, jul 1995.

[8] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, nov. 1987.

[9] Harald Gall and René Klösch. Finding objects in procedural programs: An alternative approach. In *Working Conference on Reverse Engineering*, pages 208–16. IEEE, IEEE Comp. Soc. Press, jul 1995.

[10] Donald A. Jackson, Keith M. Somers, and Harold H. Harvey. Similarity Coefficients: Measures of Co-occurence and Association or Simply Measures of Occurence ? *The American Naturalist*, 133(3):436–453, March 1989.

[11] Thomas Kunz and James P. Black. Using automatic process clustering for design recovery and distributed debugging. *IEEE Transaction on Software Engineering*, 21(6):515–527, jun 1995.

[12] A. De Lucia, G.A. Di Lucca, A.R. Fasolino, P. Guerra, and S. Petruzzelli. Migrating legacy systems towards object-oriented platforms. In Mary Jean Harrold and Guieppe Visaggio, editors, *International Conference on Software Maintenance, ICSM'97*, pages 122–29. IEEE, IEEE Comp. Soc. Press, oct 1997.

[13] P. Lutsky. Automatic testing by reverse engineering of software documentation. In *Working Conference on Reverse Engineering*, pages 8–12. IEEE, IEEE Comp. Soc. Press, jul 1995.

[14] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance, ICSM'96*, pages 244–53. IEEE, IEEE comp. soc. press, nov. 1996.

[15] Philip Newcomb and Gordon Kotik. Reengineering procedural into object-oriented systems. In *Working Conference on Reverse Engineering*, pages 237–49. IEEE, IEEE Comp. Soc. Press, jul 1995.

[16] Sukesh Patel, William Chu, and Rich Baxter. A Measure for Composite Module Cohesion. In *14$^{th}$ International Conference on Software Engineering*. ACM SIGSoft/IEEE Comp. Soc. Press, 1992.

[17] Harry M. Sneed. Object-oriented cobol recycling. In *Working Conference on Reverse Engineering*, pages 169–78. IEEE, IEEE Comp. Soc. Press, nov 1996.

[18] Bruce W. Weide, Wayne D. Heym, and Joseph E. Hollingsworth. Reverse engineering of legacy code exposed. In *International Conference on Software Engineering, ICSE'95*, pages 327–331. IEEE, IEEE Comp. Soc. Press, 1995.

[19] Steven Woods and Qiang Yang. The program understanding problem: Analysis and a heuristic approach. In *International Conference on Software Engineering, ICSE'96*. IEEE, IEEE Comp. Soc. Press, 1996.

[20] Alexander S. Yeh, David R. Harris, and Howard B. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Working Conference on Reverse Engineering*, pages 227–36. IEEE, IEEE Comp. Soc. Press, jul 1995.