# Identifying Objects using Cluster and Concept Analysis*

Arie van Deursen, Tobias Kuipers
*CWI, P.O. Box 94079*
*1090 GB Amsterdam, The Netherlands*
*http://www.cwi.nl/∼{arie,kuipers}/, {arie,kuipers}@cwi.nl*

**Abstract**
Many approaches to support (semi-automatic) identification of objects in legacy code take the data structures as starting point for candidate classes. Unfortunately, legacy data structures tend to grow over time, and may contain many unrelated fields at the time of migration. We propose a method for identifying objects by semi-automatically restructuring the legacy data structures. Issues involved include the selection of record fields of interest, the identification of procedures actually dealing with such fields, and the construction of coherent groups of fields and procedures into candidate classes. We explore the use of cluster and concept analysis for the purpose of object identification, and we illustrate their effect on a 100,000 LOC Cobol system. Furthermore, we use these results to contrast clustering with concept analysis techniques.

## 1 INTRODUCTION

In 1976, Belady and Lehman formulated their *Laws of Program Evolution Dynamics* [1]. First, a software system that is used will undergo continuous modification. Second, the unstructuredness (entropy) of a system increases with time, unless specific work is done to improve the system's structure. One possible way of doing this is to migrate software systems to object technology. Object orientation is advocated as a way to enhance a system's correctness, robustness, extendibility, and reusability, the key factors affecting software quality [14].

The migration of legacy systems to object orientation, however, is no mean task. A first, less involved, step includes merely the identification of candidate objects in a given legacy system. The literature reports several systematic approaches to object identification, some of which can be partially automated. (In Section 2 we provide a summary). There are several problems, however, with the application of these approaches to actual systems.

1. Legacy systems greatly vary in source language, application domain, database system used, etc. It is not easy to select the identification approach best-suited for the legacy system at hand.
2. It is impossible to select a *single* object identification approach, since legacy systems typically are heterogeneous, using various languages, database systems, transaction monitors, and so on.
3. There is limited experience with actual object identification projects, making it likely that new migration projects will reveal problems not encountered before.

Thus, when embarking upon an object identification project, one will have to select and compose one's own blend of object identification techniques. Moreover, during the project, new problems will have to be solved. This is exactly what happened to us when we tried to construct an object-oriented redesign of *Mortgage*, a real life legacy Cobol system.

For many business appliations written in Cobol, the data stored and processed represent the core of the system. For that reason, the data records used in Cobol programs are the starting point for many object identification approaches (such as [4, 15, 8]).

Object identification typically consists of several steps: (1) identify legacy records as candidate classes; (2) identify legacy procedures or programs as candidate methods; (3) determine the best class for each method via some form of cluster analysis [11]. This approach gives good results in as far as the legacy record structure is adequate. In our case study, however, records consisted of up to 40 fields. An inspection of the source code revealed that in the actual use of these records, many of the fields were entirely unrelated. Making this record into a single class would lead to classes with too many unrelated attributes.

In this paper, we report on our experience with the application of some of the techniques proposed for object identification, most notably cluster and concept analysis, to *Mort-*

*This work was sponsored in part by bank ABN AMRO, software house Roccade, and the Dutch *Ministerie van Economische Zaken* (Department of Commerce) via SENTER Project #ITU95017 "SOS Resolver".

*To appear in Proceedings 21st International Conference on Software Engineering, ICSE-99, ACM, 1999*

*gage*. Moreover, we discuss in full detail how the unrelated-record-fields problem – not covered by any of the existing object identification approaches – can be addressed in general. Our approach consists of clustering record fields into coherent groups, based on the actual *usage* of these fields in the procedural code. We not only use traditional *cluster analysis* [10, 11] for this, but also the recently proposed *concept analysis* [18, 12].

The principal new results of this paper include:

- A proposal for usage-based record structuring for the purpose of object identification;
- Significant practical experience with the use of cluster and concept analysis for object identification;
- A discussion of a number of problems (and solutions) involving the use of cluster and concept analysis in general;
- A comparison of the use of cluster and concept analysis for the purpose of object identification.

## 2 RELATED WORK

A typical approach to finding classes in legacy code is to identify procedures and global variables in the legacy, and to group these together based on attributes such as use of the same global variable, having the same input parameter types, returning the same output type, etc. [16, 13, 3, 17]. A unifying framework discussing such *subsystem classification techniques* is provided by Lakhotia [11].

Unfortunately, many of these approaches rely on features such as scope rules, return types, and parameter passing, available in languages like Pascal, C, or Fortran. Many data-intensive business programs, however, are written in languages like Cobol that do not have these features. As a consequence, these class extraction approaches have not been applied successfully to Cobol systems, as was also observed by Cimitile *et al.* [4].

Other class extraction techniques have been developed specifically with languages like Cobol in mind. They take specific characteristics into account, such as the close connection with databases.

Newcomb and Kotik [15] take all level 01 records as a starting point for classes. They then proceed to map similar records to single classes, and find sections that can be associated as methods to these records. Their approach exhibits a high level of automation, and, as a consequence, results in an object-oriented program that stays close to the original Cobol sources.

Fergen *et al.* [8] describe the MOORE tool, which analyses Cobol-85 code, and provides the engineer with a set of *class proposals*. All records are given a *weight*, which indicates the number of references made to that record. No attempt is made at splitting up large records into smaller structures. Proposals for methods consist of Cobol paragraphs which use or modify one of the record fields, again ranked by the weight of the fields in that paragraph. To reduce the total number of classes, every time a new candidate class is found, a numeric *similarity* measure is used to see whether already existing classes can be used to build this new candidate class.

De Lucia *et al.* [5, 4] describe the ERCOLE paradigm for migrating RPG programs to object-oriented platforms. It consists of several steps, one of which is "abstracting an object-oriented model." This step is centered around the persistent data stores. Batch programs, subroutines, or groups of call-related subroutines are candidate methods. Data stores and methods are combined in such a way that certain object-oriented design metrics get optimal values.

Sneed and Nyáry [20] present a tool, OBJECT-REDOC, that can be used to derive documentation automatically from legacy sources. The documentation itself is "object-oriented", in that it takes an object-oriented view on the legacy system. Sneed also provides a systematic method, REORG, to transform Cobol programs to object-oriented frames in a largely manual manner [19].

Tan and Ling [23] present a domain-specific approach to reengineering data-intensive business programs. They propose the use of an *augmented object model*, which is an extension of the object modeling technique OMT. Their model recovery procedure takes constants, user inputs, retrieved and updated database records, and user outputs as its starting point. However, they make no attempt at splitting up records in smaller structures.

Wiggerts *et al.* [25] describe three different *scenarios* for object identification. Their *function*-driven scenario takes legacy *functionality* (subsystems performing a certain task) as starting point for class extraction. The *data*-driven approach starts by searching for (persistent) data elements, which are likely to describe business entities. The *object*-driven approach, finally, does not start from the legacy system itself, but starts by building an object model of the application domain.

## 3 FIELD AND PROGRAM SELECTION

Legacy systems contain data and functionality that are useful in a given application domain. Unfortunately, the legacy system also contains a significant amount of code of a technical nature, closely tied to the implementation language, operating system, database management system, etc. When migrating legacy systems to object technology, such technical code is of significantly less interest than the domain-related code, for example because the object-oriented platform is likely provide facilities for dealing with the technicalities in an entirely different manner.

Therefore, a first important step in any object identification activity must be to filter the large number of programs, pro-

cedures, records, variables, databases, etc., present in the legacy system.

One of the main selection criteria will be whether a legacy element is domain-related or implementation-specific. This is a criterion that is not easy to derive from structural code properties alone. Consequently, this step may require human interaction, in order to take advantage of domain knowledge, application knowledge, systematic naming conventions, meaningful identifiers, comments, etc.

In many cases, though, structural code properties will be able to provide a meaningful selection of legacy data elements and procedures. Selection criteria to be used may include the use of metrics, such as requiring a McCabe complexity metric between a given minimum and maximum as discussed in [2]. Others may include the classification of variables, for example according to the *type* they belong to [7] or according to whether a variable is used to represent data obtained from persistent data stores [4].

Our own experience with selecting domain-related data and functionality is described in [6]. In this paper, we will use two guidelines, one for selecting data elements and one for selecting programs. These helped to find objects in our *Mortgage* case study, and we expect them to work well for other systems too.

First, in Cobol systems the *persistent data stores* (following the terminology of [4]) contain the essential business data. Hence, the selection to be made on all records in a Cobol program is to restrict them to those written to or read from file. This selection can be further improved by taking the CRUD (Create, Read, Update, Delete) matrix for the system into account. Threshold values can be given to select those databases that are read, updated, deleted, or written by a minimal or maximal number of different programs.

Second, it is important to select the programs or procedures containing domain-related functionality. An analysis of the *program call graph* can help to identify such programs. First, programs with a high *fan-out*, i.e., programs calling many different programs, are likely to be control modules, starting up a sequence of activities. Second, programs with a high *fan-in*, being called by many different programs, are likely to contain functionality of a technical nature, such as error handling or logging. Eliminating these two categories reduces the number of programs to deal with. In many cases, the remaining programs are those containing a limited, well described functionality.

## 4 CLUSTER ANALYSIS

The goal of this paper is to identify groups of record fields that are related functionally. Cluster analysis is a technique for finding related items in a data-set. We apply cluster analysis to the usage of record fields throughout a Cobol system, based on the hypothesis that record fields that are related in the implementation (are used in the same program) are also related in the application domain.

| | $P_1$ | $P_2$ | $P_3$ | $P_4$ |
|---|---|---|---|---|
| NAME | 1 | 0 | 0 | 0 |
| TITLE | 1 | 0 | 0 | 0 |
| INITIAL | 1 | 0 | 0 | 0 |
| PREFIX | 1 | 0 | 0 | 0 |
| NUMBER | 0 | 0 | 0 | 1 |
| NUMBER-EXT | 0 | 0 | 0 | 1 |
| ZIPCD | 0 | 0 | 0 | 1 |
| STREET | 0 | 0 | 1 | 1 |
| CITY | 0 | 1 | 0 | 1 |

Table 1: The usage matrix that is used as input for the cluster analysis

| | N | T | I | P | N | NE | Z | S | C |
|---|---|---|---|---|---|---|---|---|---|
| N | 0 | | | | | | | | |
| T | 0 | 0 | | | | | | | |
| I | 0 | 0 | 0 | | | | | | |
| P | 0 | 0 | 0 | 0 | | | | | |
| N | $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ | 0 | | | | |
| NE | $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ | 0 | 0 | | | |
| Z | $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ | 0 | 0 | 0 | | |
| S | $\sqrt{3}$ | $\sqrt{3}$ | $\sqrt{3}$ | $\sqrt{3}$ | 1 | 1 | 1 | 0 | |
| C | $\sqrt{3}$ | $\sqrt{3}$ | $\sqrt{3}$ | $\sqrt{3}$ | 1 | 1 | 1 | $\sqrt{2}$ | 0 |

Table 2: The distance matrix from Table 1

In this section we will first give a general overview of the cluster analysis techniques we used. Then we give an overview of the cluster analysis experiments we performed. We end the section with an assessment of our cluster experiments and the usage of cluster analysis for object identification in general.

**OVERVIEW**

We will explain the clustering techniques we have used by going through the clustering of an imaginary Cobol system. This system consists of four programs, and uses one record containing nine fields. The names of these fields are put into the set of cluster items. For each of the variables in the set, we determine whether or not it is used in a particular program. The result of this operation is the matrix of Table 1. Each entry in the matrix shows whether a variable is used in a program (1) or not (0).

*DISTANCE MEASURES*

Because we want to perform cluster analysis on these data, we need to calculate a distance between the variables. If we see the rows of the matrix as vectors, then each variable occupies a position in a four dimensional space. We can now calculate the Euclidean distance between any two variables.

If we put the distances between any two variables in a matrix, we get a so-called *distance* (or *dissimilarity*) matrix. Such a distance matrix can be used as input to a clustering algorithm. The distance matrix for Table 1 is shown in Table 2. Note that any relation the variables had with the programs $P_1, \ldots, P_4$ has become invisible in this matrix.

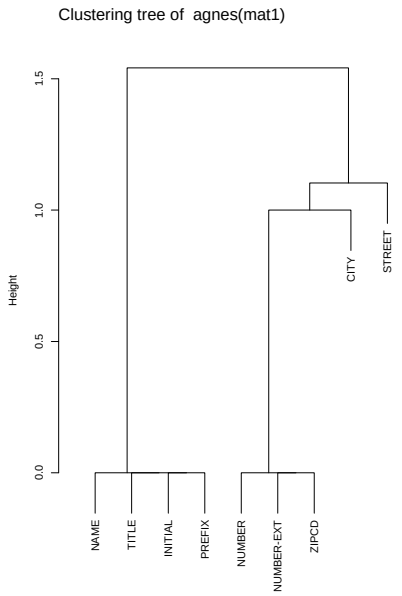



Figure 1: The resulting clustering from Table 2

An overview of different distance calculations for clustering can be found in [24].

*AGGLOMERATIVE CLUSTERING*

We use an agglomerative hierarchical clustering algorithm (AGNES, from [10]). This algorithm starts by putting each element in its own cluster, and then proceeds by creating new clusters that contain two (or more) clusters that are closest to one another. Finally, only one cluster remains, and the algorithm terminates. All intermediate clusterings can be seen as branches on a tree, in a dendrogram. Figure 1 shows the dendrogram that results from clustering the data in Table 1.

The actual clusters found by this algorithm are identified by drawing a horizontal line through the dendrogram, at a user defined height. In our example here, that line would typically be drawn at height 1.3, thus producing two clusters. The first cluster contains NAME, TITLE, INITIAL, and PREFIX. The second contains NUMBER, NUMBER-EXT, ZIPCD, CITY, and STREET. These clusters are likely candidates to become classes, containing the named fields as their member variables.

*EXPLANATION OF DENDROGRAM*

In Figure 1, the axis labelled "height" shows the relative distance the clusters have from each other. The variables NAME, TITLE, INITIAL, and PREFIX have a relative distance of zero (see Table 2), and thus form one cluster. We will call this cluster $c_1$. NUMBER, NUMBER-EXT and ZIPCD also have distance zero. We will call this cluster $c_2$. No other clusters with members that have distance 0 exist.

The clustering algorithm uses "average linkage" to measure the distance between two clusters. This means that the distance between two clusters is the average of the distances between all nodes of the one cluster, and all nodes of the other cluster. (See [24] for a discussion of this and other linkage methods.) Using this linkage method, the closest element to cluster $c_2$ is either CITY, or STREET. They both have a distance of 1 to $c_2$. The clustering algorithm nondeterministically chooses one of CITY or STREET. In our case it chooses CITY. $c_2$ and CITY together form cluster $c_3$.

The element closest to $c_3$ is STREET. It has a distance of $\sqrt{2}$ to CITY, and a distance of 1 to all elements of $c_2$. So, on average, the distance between STREET and $c_3$ is $\frac{3+\sqrt{2}}{4} \approx 1.1$. This new cluster we will call $c_4$.

Now, only two clusters remain: $c_1$ and $c_4$. The distance between these two clusters is $\frac{4\times(3\times\sqrt{2}+2\times\sqrt{3})}{4\times5} \approx 1.54$.

## EXPERIMENTAL TESTBED

The input data for our cluster experiments was generated from Cobol source code, using lexical analysis tools. The data from these tools was fed into a relational database. We wrote a tool to retrieve the data from the database, and to format it for our cluster tools. The source code was from *Mortgage*, a 100.000 LOC Cobol system from the banking area. It uses VSAM files for storing data. The toolset used for the generation of data, and the architecture of those tools is described in more detail in [6]. The *Mortgage* system is described in more detail in [6, 25].

For our cluster experiments we used S-PLUS, a statistical analysis package from MathSoft. The cluster algorithms described in [10] are implemented as part of S-PLUS.[1]

All experiments were performed on a SGI O2 workstation.

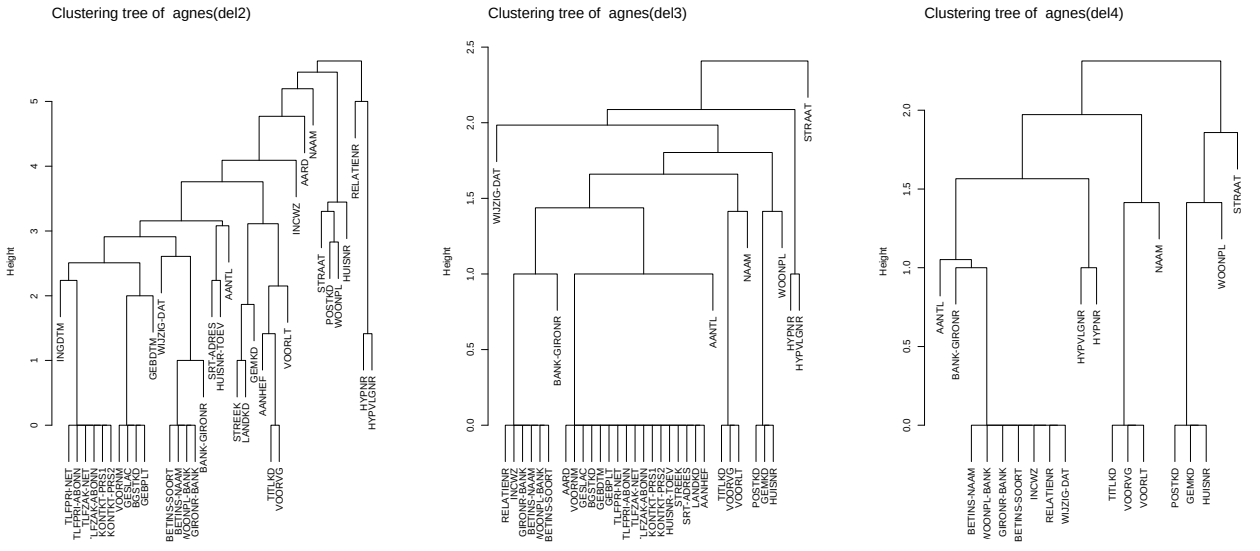
## EXPERIMENTS

As already described in Section 3, we selected a number of variables and programs from *Mortgage* to perform our cluster experiments on. In this section we will describe our main experiment, which was executed in three steps. The results of the clustering experiments are shown in Figure 2. As stated before, we are looking for clusters of functionally related record fields. In order to validate the use of cluster analysis for this purpose, we need to validate the clusters found. We have asked engineers with an in-depth knowledge of the system to validate the clusters for us.

The (variable) names mentioned in the dendrograms of Figure 2 are in Dutch. We will translate the names as we explain the three dendrograms of that figure.

1. We restricted the variables to be clustered to only those occurring in the three main records of *Mortgage*. This led to the dendrogram of Figure 2(a). There are a number of groups that seem meaningful, such as STRAAT, POSTKD, WOONPL and HUISNR (street, zip code, city and street number), or the cluster containing STREEK,

[1]The implementation is available from http://win-www.uia.ac.be/u/statis/programs/clusplus_readme.html




(a) Clustering using variables from three main records of *Mortgage*.

(b) As Figure 2(a), but restricted to the relevant programs (with low fan-in and fan-out).

(c) As Figure 2(b), but without the programs which use all variables from the three records.

Figure 2: Sequence of more and more refined clustering

LANDKD, and GEMKD (region, country code, county code). In short, this dendrogram does illustrate which variables are used together frequently, and which could therefore be grouped together.

Unfortunately, there are also a number of fields with a position that is not so easy to explain. These are in particular the ones with a "higher" position, such as INCWZ, AARD, NAAM or AANTL (payment, kind, name, and occurrence). Also, the grouping of contact persons (KONTKT-PERS) with telephone numbers (everything starting with TLF) is unclear.

2. The next step is to restrict the number of programs involved. Figure 2(b) shows the clustering results when only programs from the group of "relevant programs" (as described in Section 3) were taken into account.

   The result is promising, and has a simpler tree structure. However, there is an unattractively large group of fields that are lumped together, which does not look very meaningful. The reason for this is that there are two programs in the group of relevant programs which use *all* variables. Therefore their discriminating capabilities in the clustering are very low.

3. We repeated the same experiment, but now without the programs which use all variables. The result is the dendrogram of Figure 2(c). This is a very satisfying cluster result.

Note that the last dendrogram contains significantly less field names than the first. This makes it easier to comprehend the clusters, but also means that we have to inspect all removed variables manually for inclusion in one (or none) of the generated clusters.

**ASSESSMENT**

We have identified two fundamental problems when using cluster analysis this way:

1. When clustering, all items end up in exactly one cluster. However, sometimes one item (one variable) is equally likely to end up in more than one cluster. For instance, two records may use the same key field. If all other fields of the records are disjoint, and are used disjointly, we end up with three clusters: one containing the fields of the first record, without the key field, one with the fields of the second record without the key field, and one with only the key field. It is unclear whether this is the most desirable result. Perhaps we would rather have two clusters, corresponding exactly to the two records. Unfortunately, as items can only occur in exactly one cluster, this is not possible using cluster analysis.

2. As we have demonstrated in our example, when we are building the cluster hierarchy, sometimes there is more than one closest cluster. Assume we have a cluster $A$, which has the same distance to both clusters $B$ and $C$ (e.g., in our example, both CITY and STREET had a distance of 1 to cluster $c_2$). The algorithm at that point chooses one, arbitrarily. Say the algorithm chooses cluster $B$, thus forming cluster $A'$. Now cluster

$A'$ has a particular distance to cluster $D$ which may be very different from the distance it had had if the algorithm had chosen $C$ and $A$ to form $A'$. If this happens near the leaves of the dendrogram, the results of an arbitrary decision can be drastic.

We have partly solved these problems as follows:

1. The fields most likely to end up in more than one cluster are fields that are used together with a lot of other fields. Or, in short, the fields that are used most often. The system we experimented with demonstrated this property. The above mentioned key field is obviously used quite often, because it uniquely identifies a record. We have overcome the restrictions of the cluster algorithm by removing these variables from our cluster set before starting the cluster analysis. This proved to be a satisfactory method.

   Automatic variable selection procedures in cluster algorithms have been proposed in the literature [9]. It is a topic of future research to incorporate these procedures in our clustering experiments.

2. We have tried to resolve the second problem by changing the distance metrics and the linkage methods between clusters. We experimented with all metrics and methods described in [24]. However, although changing these parameters indeed resulted in different clusters, it did not necessarily result in *better* clusters. The problem here is that it often is unclear which of the choices is the better choice, and indeed the choice *is* arbitrary. What sometimes is clear is that a particular sequence of choices is to be preferred above another sequence. We have not tried to incorporate this notion into our cluster algorithm. This would probably require some type of backtracking mechanism, or a multiple pass algorithm, and is a topic of further research.

In conclusion we can say that cluster analysis can be used for restructuring records, given a number of restrictions. First, the number of fields to be clustered cannot be too large. Second, the fields to be clustered should be occurring selectively in the system (i.e., they should not be omnipresent fields, for these generate noise). Finally, there needs to be some external way to validate the clustering.

## 5 CONCEPT ANALYSIS

Recently, the use of mathematical *concept analysis* has been proposed as a technique for analyzing the modular structure of legacy software [12, 18, 21, 22]. As with cluster analysis, we use concept analysis to find groups of record fields that are related in the application domain.

Concept analysis and cluster analysis both start with a table indicating the *features* of a given set of *items*. Cluster analysis then partitions the set of items in a series of disjoint clusters, by means of a numeric distance measure between items indicating how many features they share.

Concept analysis differs in two respects. First, it does not group items, but rather builds up so-called *concepts* which are maximal sets of items sharing certain features. Second, it does not try to find a single optimal grouping based on numeric distances. Instead it constructs *all* possible concepts, via a concise lattice representation.

As we will see in the next paragraphs, these two differences can help to solve the two problems with clustering discussed in the previous section. In this section, we will first explain the basics of concept analysis. Then we will discuss its application to our *Mortgage* case study in full detail, followed by a comparison with the clustering results.

### BASIC NOTIONS

We start with a set $M$ of *items*, a set $F$ of *features*,[2] and a *feature table* (relation) $T \subseteq M \times F$ indicating the features possessed by each item. If we reuse the data of Table 1 as running example, the items are the field names, the features are usage in a given program, and the feature table corresponds to the matrix entries having value 1.

For a set of items $I \subseteq M$, we can identify the *common features*, written $\sigma(I)$, via:

$$\sigma(I) = \{ f \in F \mid \forall i \in I : (i, f) \in T \}$$

For example, $\sigma(\{\texttt{ZIPCD}, \texttt{STREET}\}) = \{P_4\}$.

Likewise, we define for $F \subseteq F$ the set of *common items*, written $\tau(F)$, as:

$$\tau(F) = \{ i \in M \mid \forall f \in F : (i, f) \in T \}$$

For example, $\tau(\{P_3, P_4\}) = \{\texttt{STREET}\}$.

A *concept* is a pair $(I, F)$ of items and features such that $F = \sigma(I)$ and $I = \tau(F)$. In other words, a concept is a maximal collection of items sharing common features. In our example,

$$(\{\texttt{NAME}, \texttt{TITLE}, \texttt{INITIAL}, \texttt{PREFIX}\}, \{P_1\})$$

is the concept of those items having feature $P_1$, i.e., the fields used in program $P_1$. All concepts that can be identified from Table 1 are summarized in Table 3. The items of a concept are called its *extent*, and the features its *intent*.

The concepts of a given table form a partial order via:

$$(I_1, F_1) \leq (I_2, F_2) \;\Leftrightarrow\; I_1 \subseteq I_2 \;\Leftrightarrow\; F_2 \subseteq F_1$$

As an example, for the concepts listed in Table 3, we see that bot $\leq c3 \leq c2 \leq$ top.

[2] The literature generally uses *object* for *item*, and *attribute* for *feature*. In order to avoid confusion with the objects and attributes from object orientation we have changed these names into items and features.

| name | extent | intent |
|---|---|---|
| top | {NAME, TITLE, INITIAL, PREFIX, NUMBER, NUMBER-EXT, ZIPCD, STREET, CITY} | $\emptyset$ |
| c1 | {NAME, TITLE, INITIAL, PREFIX} | $\{P_1\}$ |
| c2 | {NUMBER, NUMBER-EXT, ZIPCD, STREET, CITY } | $\{P_4\}$ |
| c3 | {STREET} | $\{P_3, P_4\}$ |
| c4 | {CITY} | $\{P_2, P_4\}$ |
| bot | $\emptyset$ | $\{P_1, P_2, P_3, P_4\}$ |

Table 3: All concepts in the example of Table 1

The subconcept relationship allows us to organize all concepts in a *concept lattice*, with *meet* $\wedge$ and *join* $\vee$ defined as

$$\begin{aligned}(I_1,F_1)\wedge(I_2,F_2) &= (I_1\cap I_2, \sigma(I_1\cap I_2)\\ (I_1,F_1)\vee(I_2,F_2) &= (\tau(F_1\cap F_2), F_1\cap F_2)\end{aligned}$$

The visualization of the concept lattice shows all concepts, as well as the subconcept relationships between them. For our example, the lattice is shown in Figure 3. In such visualizations, the nodes only show the "new" items and features per concept. More formally, a node is labelled with an item *i* if that node is the *smallest* concept with *i* in its extent, and it is labelled with a feature *f* if it is the *largest* concept with *f* in its intent.

The concept lattice can be efficiently computed from the feature table; we refer to [12, 18, 21, 22] for more details.

**EXPERIMENTAL TESTBED**

To perform our concept analysis experiments, we reused the Cobol analysis architecture explained in Section 4. The analysis results could be easily fed into the `concept` tool developed by C. Lindig from the University of Braunschweig.[3] We particularly used the option of this tool to generate input for the graph drawing package `graphplace` in order to visualize concept lattices.

**EXPERIMENTS**

We have performed several experiments with the use of concept analysis in our *Mortgage* case study. As with clustering, the choice of items and features is a crucial step in concept analysis. The most interesting results were obtained by using exactly the same selection criteria as discussed in Section 3: the items are the fields of the relevant data records, and the programs are those with a low fan-in and fan-out. The results of this are shown in Figure 4, which shows the concept lattice for the same data as those of the dendrogram of Figure 2(b). In order to validate the use of concept analysis, we need to validate the results of the concept analysis. Again, these results were validated by systems experts.

In Figure 4 each node represents a concept. The items (field names) are names written below the concept, the features

[3]The `concept` tool is available from http://www.cs.tu-bs.de/softech/people/lindig/.

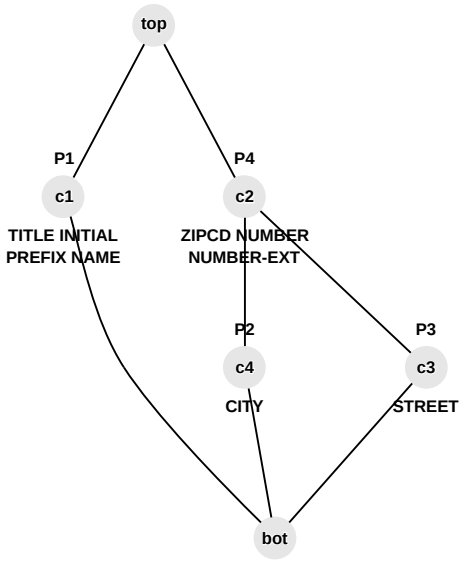



Figure 3: Lattice for the concepts of Table 3

(programs using the fields) are written as numbers above the concept. The lattice provides insight in the organization of the *Mortgage* legacy system, and gives suggestions for grouping programs and fields into classes.

The row just above the bottom element consists of five separate concepts, each containing a single field. As an example, the leftmost concept deals with *mortgage numbers* stored in the field `MORTGNR`. With it is associated program `19C`, which according to the comment lines at the beginning of this program performs certain checks on the validity of mortgage numbers. This program *only* uses the field `MORTGNR`, and no other ones.

As another example, the concept `STREET` (at the bottom right) has three different programs directly associated with it. Of these, `40` and `40C` compute a certain standardized extract from a street, while program `38` takes care of standardizing street names.

If we move up in the lattice, the concepts become larger, i.e., contain more items. The leftmost concept at the second row contains *three* different fields: the *mortgage sequence number* `MORTSEQNR` written directly at the node, as well as the two fields from the lower concepts connected to it, `MORTGNR` and `RELNR`. Program `09` uses all three fields to search for full mortgage and relation records.

Another concept of interest is the last one of the second row. It represents the combination of the fields `ZIPCD` (zip code), `HOUSE` (house number), and `CITYCD` (city code), together with `STREET` and `CITY`. This combination of five is a separate concept, because it actually occurs in four different pro-

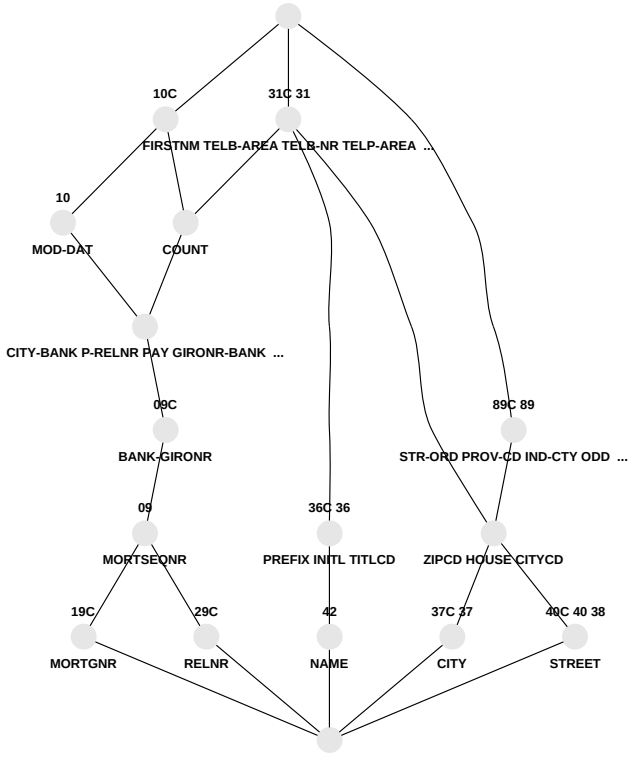



Figure 4: Concept lattice showing how persistent fields are used in programs in the *Mortgage* case study.

grams (`89C`, `89`, `31C`, `31`). However, there are no programs that *only* use these variables, and hence this concept has no program associated with it.

The largest concepts reside in the top of the lattice, as these collect all fields of the connected concepts lower in the lattice. For example, the concept with programs `31` and `31C` consists of a range of fields directly attached with it (`FIRSTNM`, ...), as well as of all those in the three downward links below it. It corresponds to almost all fields of one particularly large record, holding the data of so-called *relations* (people and companies that play a role when a mortgage is set up). These fields are then processed by programs `31` and `31C`. Only one field, `MOD-DAT` (modification date), is part of that *relations* record but not used in `31` and `31C`.

Another large concept of interest is the one with programs `89C` and `89`. The fields in this concept all come from the Dutch *zip code book*, holding data for all Dutch addresses and their zip codes. As can be seen from Figure 4, the fields of this concept are largely disjoint with those of the *relations* concept (with programs `31` and `31C`). However, these two concepts also share five fields, namely those of the `ZIPCD` concept. These fields can be used (in various combinations) as the lookup key for the zip code book.

## 6 CLUSTERING AND CONCEPTS COMPARED

The application of both concept and cluster analysis to *Mortgage* highlights the differences listed below. From them, we conclude that concept analysis is more suitable for object identification than cluster analysis.

**Multiple partitionings** Having a hierarchy of clusterings rather than a single partitioning result, is attractive as it allows one to select the most suitable clustering.

At first sight, a dendrogram seems to provide exactly such a hierarchy. Unfortunately, as we have seen in Section 4, the actual clusters built in the final iterations of an agglomerative analysis strongly depend on clustering decisions made earlier in the analysis. It is certainly not the case that a dendrogram shows all possible clusterings.

Concept analysis, by contrast, shows *all* possible groupings that are meaningful given the feature table. In our experience, this is more helpful for the engineer trying to understand a legacy system.

**Items in multiple groups** With cluster analysis, the result is a *partitioning*, i.e., each item is placed in exactly one cluster. In some cases, however, it is important to group items in multiple clusters. For our type of experiments, for example, database *key* fields may occur in multiple records: once as *primary* key, and in potentially multiple other records as *foreign* key.

With concept analysis, unlike clustering, this is possible. In our experiments, key fields occur as separate concepts, with separate upward links to those concepts using them as either primary or foreign key. In Figure 4, the zip code concept is an example of such a key concept.

Moreover, if concept analysis is used, it still is possible to obtain a partitioning, following an algorithm discussed in [18].

**Features and Clusters** For class extraction purposes, it is important to understand which features were responsible for the creation of certain clusters. With cluster analysis, this is virtually impossible, as the analysis is solely based on the distance matrix (see Table 2), in which no feature appears.

The absence of features also makes dendrograms more difficult to interpret than concept lattices. For example, in Figure 4 it is clear that program `10` is responsible for the special status of `MOD-DAT`, but in Figure 2(b) it is not at all obvious why `STRAAT` (street) appears at the top of the dendrogram.

**Selection of input data** The appropriate selection of input data stronly affects the results of both cluster and concept analysis. Cluster analysis turns out to be very sensitive to items that possess *all* features. As a result, we have derived two extra selection steps for cluster analysis: Remove programs that use all fields from the input data, and remove

record fields that are used in all programs from the input data.

Concept anlaysis is also sensitive to the selection of input data, but less so: therefore, we were able to derive the concept lattice of Figure 4 from the data used for the dendrogram in Figure 2(b), rather than from the more restricted dataset used in Figure 2(c).

## 7 OBJECT IDENTIFICATION

The final object identification step is to use the cluster and concept analysis results to build object-oriented classes. Although some degree of automation may be possible for this step, meaningful classes can be expected only if it is done interactively by a software engineer equipped with experience in object-oriented design as well as some knowledge of the application domain and the legacy system. The role of cluster and concept analysis, then, is to reduce the overwhelming number of 100,000 lines of code to a number of high-level design decisions.

When using cluster analysis, the engineer will have to decide at which height the clusters are to be chosen in a given dendrogram. This determines how many clusters exist, how large they are, and what is contained in them. Each cluster represents a candidate class, having the fields in the cluster as its attributes. The cluster hierarchy present in a dendrogram also gives pointers for relations between the classes. If a large cluster $c$ is obtained by merging clusters $c_1, \ldots, c_n$, the corresponding class $c$ will typically be composed from the classes for $c_1, \ldots, c_n$ via aggregation ($c$ will have $n$ attributes for fields of type $c_1, \ldots, c_n$). In some cases, inheritance or association may be more appropriate, but the dendrogram itself provides no clues for making this decision. Cluster analysis provides no information on which methods to attach to each of the classes identified.

When using concept analysis, the engineer can take advantage of the presence of the programs (as features) in the lattice. An important use of the lattice is as a starting point for acquiring understanding of the legacy system. As illustrated by the discussion of the *Mortgage* experiment in Section 5, the engineer can browse through the lattice, and use it to select programs at which to look in more detail.

Each concept is a candidate class. The smallest concept introducing a field corresponds to the class having that field as attribute. The largest concept with a given program as feature corresponds to the class with that program attached as method to it. This is reflected in the way the concepts are labeled with items and features in the concept lattice. Classes close to the bottom are the smallest classes (containing few attributes).

The subconcept relationship corresponds to class relations. Typically, a class for a concept $c$ is composed via aggregation from the classes of the subconcepts of $c$. Alternatively, if a concept $c$ has a subconcept $c'$, $c$ may be composed from $c'$ via inheritance. As an example, the concept with field `NAME` (and program `42`) in Figure 4 deals with names of persons. A natural refinement of this class is the concept above it, which extends a person's name with his prefixes, initials, and title code. Independent "columns" in the concept lattice correspond to separate class hierarchies.

A final question of interest is whether the classes found this way are "good" classes. For *Mortgage*, an independent, manually developed, object-oriented redesign exists (which is partly described by [25]). A good semi-automatic approach should get as close as possible to this redesign. The lattice of Figure 4 does not yield the complete redesign, but the concepts in the lattice constitute the core classes of the independent redesign. One difference is that certain large "container" classes are not present in the lattice. A second difference is that in the redesign domain knowledge was used to further refine certain classes (for example, a separate "bank address" class was included). However, this separation was not explicitly present in the legacy system. For that reason, it was not included in the concept lattice, as this only serves to show how fields are actually being used in the legacy system.

## 8 CONCLUDING REMARKS

In this paper we have studied the object identification step of combining legacy data structures with legacy functionality. We have used both cluster and concept analysis for this step. Concept analysis solves a number of problems encountered when using cluster analysis.

Of utmost importance with both concept and cluster analysis is the appropriate selection of the items and features used as a starting point, in order to separate the technical, platform-specific legacy code from the more relevant domain-related code. The selection criteria we used are discussed in Section 3.

When searching for objects in data-intensive systems (which is the typically the case with Cobol systems), records are a natural starting point. We have argued that it is first necessary to decompose the records into smaller ones, and we have proposed a method of doing so by grouping record fields based on their actual usage in legacy programs.

We have used this grouping problem to contrast cluster analysis with concept analysis. We identified the following problems with cluster analysis (see Section 6): (1) cluster analysis only constructs *partitionings*, while it is often necessary to place items in multiple groups; (2) a dendrogram only shows a subset (a hierarchy) of the possible partionings, potentially leaving out useful ones; (3) a dendrogram is difficult to explain, as it is based on numeric distances rather than actual features; (4) cluster analysis tends to be sensitive to items possessing *all* features.

These limitations are inherent to clustering, and independent of the distance measures chosen, or the sort of items used to cluster on.

These problems are dealt with in a better way by concept analysis, making it therefore more suitable for the purpose of object identification. Concept analysis finds all possible combinations, and is not just restricted to partitionings. Moreover, the features are explicitly available, making it easier to understand why the given concepts emerge.

**ACKNOWLEDGMENTS**

We thank the members of the *Object and Component Discovery* Resolver task group: Hans Bosma, Erwin Fielt, Jan-Willem Hubbers, and Theo Wiggerts. Finally, we thank Andrea De Lucia, Jan Heering, Paul Klint, Christian Lindig, and the anonymous referees for commenting on earlier versions of this document.

**REFERENCES**

[1] BELADY, L. A., AND LEHMAN, M. M. A model of large program development. *IBM Systems Journal 15*, 3 (1976), 225–252.

[2] CALDIERA, G., AND BASILI, V. R. Identifying and qualifying reusable software components. *IEEE Computer* (February 1991), 61–70.

[3] CANFORA, G., CIMITILE, A., AND MUNRO, M. An improved algorithm for identifying objects in code. *Software—Practice and Experience 26*, 1 (1996), 25–48.

[4] CIMITILE, A., DE LUCIA, A., DI LUCCA, G. A., AND FASOLINO, A. R. Identifying objects in legacy systems using design metrics. *Journal of Systems and Software 44*, 3 (1999), 199–211.

[5] DE LUCIA, A., DI LUCCA, G. A., FASOLINO, A. R., GUERRA, P., AND PETRUZZELLI, S. Migrating legacy systems towards object-oriented platforms. In *International Conference on Software Maintenance; ICSM'97* (1997), IEEE Computer Society, pp. 122–129.

[6] DEURSEN, A. V., AND KUIPERS, T. Rapid system understanding: Two COBOL case studies. In *International Workshop on Program Comprehension* (1998), IEEE Computer Society, pp. 90–97.

[7] DEURSEN, A. V., AND MOONEN, L. Type inference for COBOL systems. In *Working Conference on Reverse Engineering; WCRE'98* (1998), IEEE Computer Society, pp. 220–230.

[8] FERGEN, H., REICHELT, P., AND SCHMIDT, K. P. Bringing objects into COBOL: MOORE - a tool for migration from COBOL85 to object-oriented COBOL. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS 14)* (1994), Prentice-Hall, pp. 435–448.

[9] FOWLKES, E. B., GNANADESIKAN, R., AND KETTENRING, J. R. Variable selection in clustering. *Journal of Classification 5* (1988), 205–228.

[10] KAUFMAN, L., AND ROUSSEEUW, P. J. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[11] LAKHOTIA, A. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software* (March 1997), 211–231.

[12] LINDIG, C., AND SNELTING, G. Assessing modular structure of legacy code based on mathematical concept analysis. In *19th International Conference on Software Engineering, ICSE-19* (1997), ACM Press, pp. 349–359.

[13] LIU, S. S., AND WILDE, N. Identifying objects in a conventional procedural language: An example of data design recovery. In *International Conference on Software Maintenance; ICSM'90* (1990), IEEE Computer Society, pp. 266–271.

[14] MEYER, B. *Object-Oriented Software Construction*, second ed. Prentice Hall, 1997.

[15] NEWCOMB, P., AND KOTTIK, G. Reengineering procedural into object-oriented systems. In *Second Working Conference on Reverse Engineering; WCRE95* (1995), IEEE Computer Society, pp. 237–249.

[16] ONG, C. L., AND TSAI, W. T. Class and object extraction from imperative code. *Journal of Object-Oriented Programming* (March–April 1993), 58–68.

[17] SCHWANKE, R. W. An intelligent tool for reengineering software modularity. In *13th International Conference on Software Engineering, ICSE-13* (1991), IEEE, pp. 83–92.

[18] SIFF, M., AND REPS, T. Identifying modules via concept analysis. In *International Conference on Software Maintenance, ICSM97* (1997), IEEE Computer Society.

[19] SNEED, H. M. Migration of procedurally oriented COBOL programs in an object-oriented architecture. In *International Conference on Software Maintenance; ICSM'92* (1992), IEEE Computer Society, pp. 105–116.

[20] SNEED, H. M., AND NYÁRY, E. Extracting object-oriented specification from procedurally oriented programs. In *Second Working Conference on Reverse Engineering; WCRE'95* (1995), IEEE Computer Society, pp. 217–226.

[21] SNELTING, G. Concept analysis — a new framework for program understanding. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)* (1998). SIGPLAN Notices 33(7).

[22] SNELTING, G., AND TIP, F. Reengineering class hierarchies using concept analysis. In *Foundations of Software Engineering, FSE-6* (1998), ACM, pp. 99–110. SIGSOFT Software Engineering Notes 23(6).

[23] TAN, H. B. K., AND LING, T. W. Recovery of object-oriented design from existing data-intensive business programs. *Information and Software Technology 37*, 2 (1995), 67–77.

[24] WIGGERTS, T. Using clustering algorithms in legacy systems remodularization. In *4th Working Conference on Reverse Engineering* (1997), IEEE Computer Society, pp. 33–43.

[25] WIGGERTS, T., BOSMA, H., AND FIELT, E. Scenarios for the identification of objects in legacy systems. In *4th Working Conference on Reverse Engineering* (1997), IEEE Computer Society, pp. 24–32.