



## Identifying objects in legacy systems using design metrics

Aniello Cimitile <sup>a,\*</sup>, Andrea De Lucia <sup>a</sup>, Giuseppe Antonio Di Lucca <sup>b</sup>, Anna Rita Fasolino <sup>c</sup>

<sup>a</sup> Faculty of Engineering, University of Sannio, Palazzo Bosco Lucarelli, Piazza Roma, 82100 Benevento, Italy

<sup>b</sup> Dipartimento di Informatica e Sistemistica, University of Naples "Federico II", Via Claudio 21, 80125 Naples, Italy

<sup>c</sup> Dipartimento di Informatica, University of Bari, Via Orabona, 4-70125 Bari, Italy

Received 1 June 1997; accepted 1 April 1998

---

### Abstract

Many organisations are migrating towards object-oriented technology. However, owing to the business value of legacy software, new object-oriented development has to be weighed against salvaging strategies. The incremental migration of procedurally oriented systems to object-oriented platforms seems to be a feasible approach, although it must be considered as risky as redevelopment. This approach uses reverse engineering activities to abstract an object-oriented model from legacy code. The paper presents a method for decomposing legacy systems into objects. The identification of objects is centred around persistent data stores, such as files or tables in the database, while programs and routines are candidates for implementing the object methods. Associating the methods to the objects is achieved by optimising selected object-oriented design metrics. The rationale behind this choice is that the object-oriented decomposition of a legacy system should not result in a poor design, as this would make the re-engineered system more difficult to maintain. © 1999 Elsevier Science Inc. All rights reserved.

---

### 1. Introduction

Object technology is widely considered as the most suitable means for moving towards open systems based on distributed client-server platforms, thus significantly reducing development and maintenance costs with respect to centralised mainframe systems. Many organisations are therefore migrating towards object-oriented technology (Graham, 1994). However, most legacy software consists of procedurally oriented systems running on mainframes. Redeveloping these systems is risky and expensive and it can take years before the functions available on the old mainframe system are replaced by a reliable and equivalent object-oriented system. The business value of legacy software makes it necessary to weigh salvaging strategies against new object-oriented development (Sneed, 1995). Re-engineering procedurally oriented systems to create object-oriented architectures seems to be a more feasible approach, although it has to be considered as risky as redevelopment (Sneed, 1996a). Migrating an existing system could be an evolutionary process (Jacobson and Lindström, 1991; De Lucia et al., 1997) requiring parts of the old system to remain temporarily in use. Coexistence between old procedural

parts and new object-oriented parts is now possible thanks to new technologies, such as the OMG's Common Object Request Broker Architecture (CORBA), that facilitate the encapsulation of existing software components into object wrappers (Dietrich et al., 1989; Sneed, 1996a).

Migrating legacy systems towards object-oriented platforms is one of the goals of the PROGRESS project, a research project on process and software re-engineering in Italian government organisations promoted by the Italian National Research Council (CNR) and the Italian National Academic Consortium (CINI) and carried out by Italian universities and research centres. This migration goal has strategic relevance because the demand for maintenance/evolution of existing systems is likely to increase rapidly in the near future as a result of major innovations in both the administrative processes and the technologies in use. Two major government initiatives that will certainly contribute to a growing demand for maintenance are:

- the creation of a nation-wide open network and the definition of an associated co-operative architecture to allow information systems (new and existing) from different organisations to be integrated;
- the redesign/reorganisation of administrative processes to improve the delivery of public services, speed up

---

\* Corresponding author. E-mail: cimitile@unina.it.

decision-making, devise new ways of realising and supplying public services.

These two initiatives are closely interconnected. The creation of a nation-wide co-operative network of information systems is crucial for process re-engineering as it will go beyond the boundaries of single government organisations to design inter-organisational processes and services.

In this context, methods and processes are needed to decompose existing systems into notable sets of components, each of which potentially implements an object. The decomposition of legacy systems has to be integrated with the definition of the new processes and data regarding the government organisations: the potential objects identified have to be submitted to a concept assignment process (Biggerstaff et al., 1994) and mapped onto a new object-oriented conceptual model of the application domain resulting from the process redesign/reorganisation activities. Each set of components corresponding to an application domain concept can be encapsulated into a different object wrapper and used within the network by new object-oriented systems through the wrapper interface. The decomposition of the legacy system is the starting point for an incremental migration strategy: each object can be re-implemented independently using new object-oriented technologies; old components can be used in their original form until the new equivalent objects guarantee an acceptable level of reliability.

This paper presents a method for recovering objects from data-centred legacy systems developed using a procedural language such as COBOL, PL1, or RPG. The identification of objects is centred around persistent data stores, such as files or tables in the database, while programs and routines are candidates for implementing the object methods. The association of the methods to the objects is achieved by optimising selected object-oriented design metrics (Chidamber and Kemerer, 1994). In particular, it is made while trying to minimise the coupling between the objects. This decision was made to prevent object-oriented decomposition of a legacy system from being detrimental to the design, and consequent maintenance, of the re-engineered system.

The paper is organised as follows. In Section 2, related work is discussed. Section 3 presents an overview of the method for identifying objects, while details about the association of services to objects, based on design metrics and dominance analysis, are given in Sections 4 and 5, respectively. Section 6 describes the experimental results and some concluding remarks are made in Section 7.

## 2. Related work

The problem of re-engineering legacy systems to create object-oriented architectures is not new (Zimmer, 1990; Jacobson and Lindström, 1991). Zimmer (1990)

outlines the need for restructuring FORTRAN programs to achieve a style related to data abstraction and object-oriented programming, and proposes a method for identifying objects by searching for cobwebs of related variables and mapping them on global invariants. Jacobson and Lindström (1991) describe different scenarios for object-oriented re-engineering, including reverse engineering and forward engineering activities, but they do not suggest a method for identifying potential objects in legacy systems.

Identifying objects in legacy systems is a particular case of the problem of clustering system components into modules (Canfora et al., 1995). Most of the methods reported in the literature for decomposing existing software systems into modules exploit relations between *syntactic units*, such as functions, procedures, variables, and types. Lakhothia (1997) provides a unified graph-based framework for expressing different subsystem clustering techniques presented in the literature, e.g. (Belady and Evangelisti, 1981; Choi and Scacchi, 1990; Hutchens and Basili, 1985; Livadas and Johnson, 1994; Ogando et al., 1994; Ong and Tsai, 1993; Müller and Uhl, 1990; Patel et al., 1992; Schwanke, 1991).

Several methods for identifying modules in legacy systems have been thought with a target object-oriented architecture in mind. Some of them exploit the relations existing between program routines<sup>1</sup> and global variables or formal/actual parameters. A common approach Liu and Wilde, 1990; Canfora et al., 1993; Ogando et al., 1994) consists of modelling these relations in the form of bipartite graphs on which strongly connected components are then identified. Variants of this approach consider relations between routines and the formal parameters modified within the routines (Livadas and Johnson, 1994), or relations between routines and the data structures whose fields are accessed by the routines (Yeh et al., 1995). An improvement (Canfora et al., 1996) consists of using a statistical function to filter out the routines that introduce coincidental and spurious connections while identifying strongly connected components.

Other approaches consider statement blocks rather than procedures as candidates for object methods (Ong and Tsai, 1993; Achee and Carver, 1994). The method proposed by Ong and Tsai (1993) consists of two steps: in the first step heuristics are used to identify object attributes by aggregating global variables, formal parameters, and actual parameters; in the second step the object methods are extracted from the blocks around the statements referencing the identified attributes. A “greedy” approach to object

---

<sup>1</sup> The generic term routine is used to refer to the primitives used in traditional programming languages (e.g. functions, procedures, sub-routines, sections, and paragraphs) to implement functional abstractions.

identification has been proposed in (Achee and Carver, 1994). The object attributes are identified starting from the set of variables used as actual parameters in routine calls. A measure of cohesion between each pair of actual parameters is computed based on their use in the same call statement. A threshold value is then used to partition the variables into disjoint sets each corresponding to the attributes of a potential object. Finally, elementary methods of each object are identified starting from the statements manipulating the object's attributes.

Concept analysis (Wille, 1981) has also recently been applied to the problem of identifying objects in legacy systems (Lindig and Snelting, 1997; Siff and Reps, 1997). Concept analysis provides a way to identify groups of *entities*<sup>2</sup> that have common *attributes*; each group constitutes a concept, while concept partitions are represented by collections of concepts that partition the set of entities (Siff and Reps, 1997). In the case of object identification, routines are mapped onto entities, while attributes such as variable usage (*uses global variable v*) (Lindig and Snelting, 1997), or type usage (*return type is t, has argument of type t, uses structure field of type t*) (Siff and Reps, 1997) are considered; a concept partition represents a possible decomposition of a system into objects. The main advantage of this approach is the possibility of using negative attributes, such as *does not use structure fields of type t* (Siff and Reps, 1997), to obtain more refined objects. Two major limitations are the need to define the relevant attributes carefully and the high number of concept partitions resulting when the method is applied to large systems.

A different approach consists of recovering an object-oriented model of the system starting from high level design documents such as structure charts and data-flow diagrams (Gall and Klösch, 1995; George and Carter, 1996) and considering relationships between the functional and data elements in the design document. Reverse engineering methods for identifying potential objects in code are integrated with a top-down analysis of the system documentation to refine the object-oriented model (Gall and Klösch, 1995). Similarly, Subramaniam and Byrne (1996) apply object-oriented analysis to the system documentation together with reverse engineering methods (Achee and Carver, 1994; Liu and Wilde, 1990) to derive an object model from existing structured FORTRAN systems.

All these methods have given good results when applied to programs written in languages such as Pascal, C, and FORTRAN, that provide variable scope rules

and routines exchanging data through parameters declared in their heading. However, unlike the method proposed in this paper, they have not been successfully experimented with in data-intensive business programs written in languages such as COBOL or RPG, that do not provide these features.

Approaches for object-oriented re-engineering COBOL programs have been proposed in the literature, based either on formal methods (Breuer et al., 1993) or on structural methods (Newcomb and Kotik, 1995; Sneed, 1993; Sneed, 1996b). Breuer et al. (1993) describe a technique for recovering formal object-oriented specifications from code as part of the REDO project (Zuylen, 1993). The reverse engineering process involves the translation of source code to an intermediate language, called UNIFORM (Zuylen, 1993), where the relationships between data are translated into logical invariants. Objects are identified starting from the data definitions in the *file section* of COBOL programs, while methods are identified by splitting the original program into *phases* (single input–output functions) and abstracting the function associated with each phase. Newcomb and Kotik (1995) present a re-engineering tool built on the top of the REFINE™ workbench (Reasoning Systems, 1990) for automatically transforming a COBOL system into a functionally comparable object-oriented system. The transformation process entails constructing a hierarchical object-oriented state machine model; redundant, duplicated, and similar data and processes are then located and abstracted into classes and methods. The main difference with respect to the method proposed by Breuer et al. (1993) is that volatile data definitions in the *working storage section* are also analysed to identify the object state variables, as well as data definitions in the *file section* of COBOL programs. Sneed (1996b) proposes a process for extracting objects from existing COBOL programs requiring human interaction to select the data items which define object attributes from the *data division* of a COBOL program; any sequence of statements acting on the attributes of an object defines a method of this object.

A common feature of these methods is that they focus on searching for objects at the program level and suggest some solutions for merging instances of the same object class identified in different programs. On the contrary, the approach proposed in this paper starts from the persistent data stores that are global to all the programs in a software system. Volatile data items are not considered for inclusion in the set of potential objects, as persistent data stores form the relevant application domain objects to be identified in a migration process. In fact, although volatile data items can produce reusable objects, performance problems can arise when re-engineering the old system, due to excessive message passing between too fine-grained objects.

<sup>2</sup> In (Wille, 1981; Lindig and Snelting, 1997; Siff and Reps, 1997) these are called objects.

### 3. A method for identifying objects in code: An overview

Several methods for identifying objects in code can be defined depending on the desired level of granularity. For example, when identifying the data structures that implement the state of an object, a software engineer may consider only persistent data stores, such as files and/or tables in the database, or also local data structures within a program. Similarly, in the identification of the methods, procedural components can be considered at different granularity levels, i.e. programs, routines, program slices, and single statements.

The choice of the granularity level is driven by the particular problem which motivated the search for objects. The research addressed in this paper focuses on the problem of migrating existing software resources, i.e. programs and persistent data, towards an object-oriented platform to allow features of the legacy software to be exploited by new object-oriented systems. Thus, the proposed approach consists of searching for coarse-grained persistent objects, instead of fine-grained volatile objects. The structure of these objects consists of a persistent data store (one or more files or relational database tables) which defines the state of the object, and a number of services (or methods) each of which may be implemented by a program in the system, a routine (in the case of COBOL programs, paragraphs or sections can be considered) or a group of related routines on the call graph, or a program slice (Weiser, 1984). From now on, the term data store will be used to refer to persistent data stores. The method consists of two sequential phases: first, identification of the data stores implementing the data structures of potential objects; then, association of services to these objects.

#### 3.1. Identifying the data structure of objects

Persistent data stores define the elements of interest and, particularly in the case of files, a refinement step is needed to solve the problems of *synonyms* and *homonyms*. Two files are synonymous if they have different names, but the same record structure, thus corresponding to the same element of the application domain. The record descriptions of these files may be different, where the differences concern different levels of record fields decomposition. Two files are homonymous if they have the same name, but a different record structure, thus corresponding to different elements of the application domain. In particular, homonyms are due to the use of one and the same file in different programs, each of which defines a different record structure for this file.

The process of identifying the object data structures can thus be decomposed into two phases. In the first phase a static analysis of the system, together with an analysis of the available documentation, is performed to

identify the persistent data stores used within the programs and their structure. The second phase consists of refining the results of the first phase by performing synonym and homonym analysis and by using the available documentation to assign an application domain concept to each of the identified objects. Of course, this task is human intensive and requires a comprehension effort from the software engineer. Synonymous files should be grouped into the same object as they may correspond to the same application domain concept. On the contrary, homonymous files should be kept separate because they represent different application domain concepts.

Generally, homonymous files are temporary files that are locally used in a program or a sub-system of the legacy application and whose state does not survive across different execution sessions. For example, these files are used in merge and/or sort operations. In some cases, temporary files are also synonymous with files whose state persists across different applications.

#### 3.2. Associating methods to objects

The approach proposed for identifying object methods consists of three sequential steps, each of which entails a different granularity level: programs, routines (or groups of routines), and slices defining object methods are identified in the three steps, respectively.

The first step attempts to associate programs to object methods by exploiting object-oriented design metrics (Chidamber and Kemerer, 1994). The rationale behind this choice is that the object-oriented decomposition of a legacy system should not mar the design, as this would harm the re-engineered system's ease of maintenance. In particular, the association is made while trying to minimise the coupling between the objects (see Section 4 for the underlying theoretical basis). Measures of the coupling between programs and persistent data stores are computed based on the accesses (*I/O* operations) of programs to data stores. The programs are associated to the data stores according to the distribution minimising the total coupling between the potential objects. Minimisation of the coupling between objects is achieved by associating each program to the data store it is most highly coupled with. However, a program is a candidate to become an object method only in the following two cases:

- the program is coupled *exclusively* to the object it has been associated, i.e., it does not access the data stores of other objects;
- the program is coupled *predominantly* to the object it has been associated, with respect to the other objects it accesses.

In the first case, the program can naturally be considered as a method of the object it accesses, while in the

second case the accesses of the program to the other objects have to be considered as messages to elementary methods implementing *I/O* operations. This is different from the case of programs which *uniformly* access the data stores of more than one object. In this case the program has to be decomposed into smaller components, each of which can be associated to a different object.<sup>3</sup>

The second step entails decomposing programs with uniform coupling into routines (or groups of routines) defining object methods. To this aim, measures of the coupling between routines and persistent data stores are computed. Clusters of routines defining object methods are then identified using dominance analysis (Hecht, 1977) of the call graph and object coupling minimisation techniques: (i) the *routine call graph* is transformed into a *call dominance tree* (Cimitile and Visaggio, 1995) and (ii) each sub-tree containing routines exclusively or predominantly coupled to the same object is a candidate to become a method of this object. Details of this step are given in Section 5.

At the end of the second step, there may still be routines uniformly coupled to different objects. In the third step, slicing techniques (Weiser, 1984; Ferrante et al., 1987; Horwitz et al., 1990) can be used to decompose, where possible, these routines into chunks of code implementing methods of the different objects. Program slicing techniques have not been investigated in this paper.

#### 4. Associating program units to objects using design metrics

The identification of object methods entails associating program components at different granularity levels to the persistent data stores implementing the data structures of potential objects. At the program and routine levels, this association is based on design metrics. From now on, the term *program unit* will refer to either programs or routines depending on the granularity level considered. This section outlines the theoretical basis for associating program units to object methods according to minimal coupling.

A software system is composed of a set  $PP$  of  $n$  program units and a set  $DD$  of  $m$  persistent data stores. The association of program units to object methods starts from a system representation model consisting of a bipartite graph, whose set of nodes is  $PP \cup DD$  and whose set of edges depicts accesses of program units to data stores. Depending on the particular association

method, the edges of the graph may be weighted: the weight may indicate the number of accesses of a program unit to a data store, for example. In this case the set of edges represents the relation  $ACCESSES \subseteq PP \times DD \times \mathcal{R}$ , where, where  $\mathcal{R}$  is the set of real numbers used to represent the weights of the edges. It is worth stressing that this relation can be obtained by static analysis of the source code.

The relation  $ACCESSES$  can be modelled by an  $n \times m$  matrix whose rows and columns correspond to program units and data stores, respectively, and whose generic entry  $\alpha_{ij}$  depends on the accesses the program unit  $i$  makes to the data store  $j$ . Association of program units to data stores can be modelled by an  $n \times m$  matrix  $X$  whose generic entry  $x_{ij}$  assumes the value 1 if the program unit  $i$  is associated to the data store  $j$ , and the value 0 otherwise. The sets of constraints that must hold can be expressed as follows:

$$\forall i, \quad 1 \leq i \leq n,$$

$$\sum_{j=1}^m x_{ij} = 1,$$

$$\forall i, j, \quad 1 \leq i \leq n, 1 \leq j \leq m,$$

$$\begin{cases} \alpha_{ij} = 0 \Rightarrow x_{ij} = 0, \\ x_{ij} = 1 \Rightarrow \alpha_{ij} > 0. \end{cases}$$

The first set of constraints states that any program unit  $i$  must be associated only to one data store  $j$ , while the second set of constraints states that a program unit can be associated to a data store if, and only if, it accesses the data store (i.e., the weight of the access is not zero).

Different strategies can be adopted to associate program units to data stores. For each strategy, there will be one or more distribution of program units to data stores that meet its requirements best. The association strategy can be modelled by an objective function  $f(X)$  on the variables  $x_{ij}$ . Any solution which minimises (or maximises) the objective function can be considered as a possible distribution. In particular, association strategies based on object-oriented design metrics can be defined. For example, the *Weighted number of Methods per Class* (WMC) measure proposed by Chidamber and Kemerer (1994) could be used to define the objective function. If  $c_i$  denotes the weight of the program unit  $i$ ,  $c_i$  being computed on the quality characteristics of the program unit (e.g., its complexity), the WMC of an object  $j$  can be expressed as

$$wmc_j = \sum_{i=1}^n c_i x_{ij}.$$

In this case, the association strategy could aim to obtain as uniform as possible a distribution of weighted program units to objects. This can be obtained by minimising the following function

<sup>3</sup> An alternative is to include the different data stores the program is uniformly coupled with in a single object. However, this decision can be made only after the program functions and the relative data stores have been understood.

$$f(X) = \max_{1 \leq j \leq m} (wmc_j) - \min_{1 \leq j \leq m} (wmc_j).$$

A uniform distribution of complexity strategy is achieved if a program complexity measure is used as program weight  $c_i$ . A particular case of this approach is the *uniform distribution of services* strategy obtained when all weights  $c_i$  equal 1. This approach privileges the associations that produce the most uniform distributions of program units to data stores. It is based on the assumption that objects with a limited number of services may be meaningless, while objects with too many services may produce scarcely cohesive or *potpourri* modules (Calliss and Cornelius, 1990). However, WMC based distribution strategies are afflicted by two main limitations:

- (i) the risk of producing several different, but not necessarily meaningful, distributions that minimise the objective function, and
- (ii) the fact that they do not consider the weight of accesses of program units to data stores in the objective function.

The method proposed in this paper overcomes both of these limitations and is based on coupling metrics between objects.

#### 4.1. Assignment based on minimum coupling between objects

Object coupling can be considered as an indicator of the quality of an object-oriented design. Excessive coupling is detrimental to modular design and prevents reuse (Chidamber and Kemerer, 1994). The more independent an object is, the more easily it can be reused in other applications. Moreover, highly coupled objects increase testing costs. This means that in good object-oriented design, coupling between objects should be kept to a minimum.

The relations that can be considered in computing the coupling between objects built around the data stores of a legacy system are mainly the accesses made by program units associated to one object to data stores of other objects. Hence, the objective function for a strategy based on the minimisation of coupling can be defined as follows

$$f(X) = \sum_{j=1}^m \sum_{i=1}^n (1 - x_{ij}) \alpha_{ij}.$$

For each object  $j$ , the function computes the  $\sum s_j = \sum_{i=1}^n (1 - x_{ij}) \alpha_{ij}$  of the costs  $\alpha_{ij}$  of the accesses made by all program units  $i$  that have not been associated to  $j$  (i.e., the value of  $x_{ij}$  is 0): the total coupling  $f(X)$  is given by the sum of the measures  $s_j$ . Each cost  $\alpha_{ij}$  can be computed as the weighted sum of the number of accesses the program unit  $i$  makes to the data

store  $j$ . Pilot experiments have to be carried out to define the weight of each I/O operation. The aim of these projects is to tune the values of the weights so as to achieve a good level of meaningfulness of the identified objects. In any case, the weights of operations such as *write*, that modify the state of a data store, should generally be higher than the weights of operations such as *read*, that do not alter the data store content. This is because each object should be responsible for the modification of its own state: a program unit containing modifications of a data store should therefore preferably be associated to the modified data store.

The problem of minimising the objective function above has a very simple solution given by the association of each program unit to the data store it is most highly coupled with. Indeed

$$\min \sum_{j=1}^m \sum_{i=1}^n (1 - x_{ij}) \alpha_{ij} = \sum_{j=1}^m \sum_{i=1}^n \alpha_{ij} - \min \sum_{j=1}^m \sum_{i=1}^n x_{ij} \alpha_{ij}.$$

Therefore, minimising the objective function above is equivalent to maximising the function

$$g(X) = \sum_{j=1}^m \sum_{i=1}^n x_{ij} \alpha_{ij},$$

whose solution is given by

$$\forall i, j, \quad 1 \leq i \leq n, \quad 1 \leq j \leq m,$$

$$\begin{cases} x_{ij} = 1 & \iff \alpha_{ij} = \max_{1 \leq k \leq m} \alpha_{ik}, \\ x_{ij} = 0 & \text{otherwise.} \end{cases}$$

A program unit  $i$  associated to a data store  $j$  is a candidate to become a method of the corresponding object if, and only if, it is coupled to this object in an exclusive or predominant way, while it has to be decomposed if it is uniformly coupled to more than one object. A simple heuristic can be used to distinguish between predominant and uniform coupling: let  $h$  and  $k$  be the two objects the program unit  $i$  is most coupled with

$$\alpha_{ih} = \max_{1 \leq j \leq m} \alpha_{ij} \quad \text{and} \quad \alpha_{ik} = \max_{1 \leq j \leq m, j \neq h} \alpha_{ij}.$$

The heuristic is based on the evaluation of the following function

$$t_i = \frac{\alpha_{ih} - \alpha_{ik}}{\alpha_{ih}}$$

and on the comparison of the value  $t_i$  with a threshold  $0 \leq \xi \leq 1$ . The program unit  $i$  is predominantly coupled to the object  $h$ , and then defines a method of this object, if and only if  $t_i \geq \xi$ . The value of the threshold  $\xi$  may depend on the type of legacy system analysed and can be tuned during an experimental phase.

## 5. Identifying clusters of routines implementing object methods

The second step in the approach presented in Section 3.2 for identifying object methods is the decomposition of programs with uniform distribution of data store accesses into smaller components, each of which may be associated as a different object method. This section describes how such programs may be decomposed: the granularity level considered in this step is the routine level. Several programming languages, including COBOL and RPG, allow a programmer to decompose a program into internal routines and invoke them through a proper call statement. For example, in the case of COBOL, a program can be structured in *sections* or *paragraphs* activated by *perform* statements.

The technique described in the previous section based on the minimisation of coupling between objects can be used to associate routines as object methods. However, although a single routine can be considered as an object method (whenever the routine is coupled exclusively or predominantly to the object), the program structure may embed clusters of routines, each of which defines a single object method. In order to identify these clusters, the technique based on the minimisation of coupling can be combined with analysis of the routine call graph (in the following simply call graph) of the program. A call graph of a program is a flow-graph  $CG = (PP, E, s)$ , where:

- $s \cup PP$  is the set of nodes,  $PP$  is the set of routines and  $s$  is the main program;
- $E$  is the set of edges and describes the activation relation on  $(s \cup PP) \times PP$ .

A call graph can be transformed into a hierarchy by using dominance analysis techniques (Hecht, 1977). The definitions of *dominance relation*, *direct dominance relation*, and *strong direct dominance relation* (Cimitile and Visaggio, 1995) are briefly recalled in Fig. 1.

The direct dominance relation defines a tree, called *call dominance tree* (Cimitile and Visaggio, 1995), which outlines the functional dependencies between routines: if a routine  $x$  dominates a routine  $y$  then each activation of  $y$  is preceded by an activation of  $x$ . The strong direct

dominance relation captures a fundamental characteristic of a typical functional dependency between two routines in a program: if a routine  $y$  is activated only by the routine  $x$ , then  $y$  implements a sub-function of the more general function defined by  $x$ . The set of edges of a dominance tree can be partitioned into two disjoint subsets: the set of edges representing the strong direct dominance relation and its complement set. Fig. 2 shows a call graph (a) and its call dominance tree (b). Edges representing strong direct dominance relationships are indicated by plain lines and edges representing direct dominance relationships by dashed lines. The call dominance tree can be used to aggregate routines into clusters implementing object methods using the following two heuristics:

1. Each sub-tree in the call dominance tree whose routine root is coupled exclusively or predominantly to the object  $j$  and whose other routines are either coupled exclusively or predominantly to the same object, or do not access the data structure of any object, generates a method of object  $j$ . This method includes all routines in the sub-tree and is represented by the sub-tree root. The tree obtained by collapsing all these sub-trees is called the *reduced dominance tree*.

2. Each sub-tree in the reduced dominance tree whose routine root is coupled exclusively or predominantly to object  $j$  generates a method of this object. Besides the sub-tree root, this method also includes all the routines corresponding to leaves in the sub-tree that are linked to the sub-tree root by strong direct dominance relation edges and coupled exclusively or predominantly to object  $j$ .

For example, if the routine  $s7$  in the call dominance tree in Fig. 2(b) is coupled exclusively or predominantly to an object, say  $x$ , and the other routines in the sub-tree rooted in  $s7$  are either coupled exclusively or predominantly to the same object or do not access any data store, then rule 1 can be applied to cluster this sub-tree into a method of object  $x$ . Similarly, if the routines  $s5$  and  $s13$  are coupled exclusively or predominantly to the same object, say  $y$ , while the routine  $s14$  uniformly accesses different objects, then rule 2 can be applied to cluster the routines  $s5$  and  $s13$  into a method of object  $y$ .

**Dominance Relation.** A routine  $x$  dominates a routine  $y$  if, and only if, all paths from the root  $s$  of the call graph to  $y$  go through  $x$ .

**Direct Dominance Relation.** A routine  $x$  directly dominates a routine  $y$  if, and only if,  $x$  dominates  $y$ ,  $x \neq y$ , and all the routines that dominate  $y$  dominate  $x$ , too.

**Strong Direct Dominance Relation.** A routine  $x$  strongly and directly dominates a routine  $y$  if, and only if,  $x$  directly dominates  $y$  and  $x$  is the only routine in the call graph that activates  $y$ .

Fig. 1. Definitions of dominance relations.

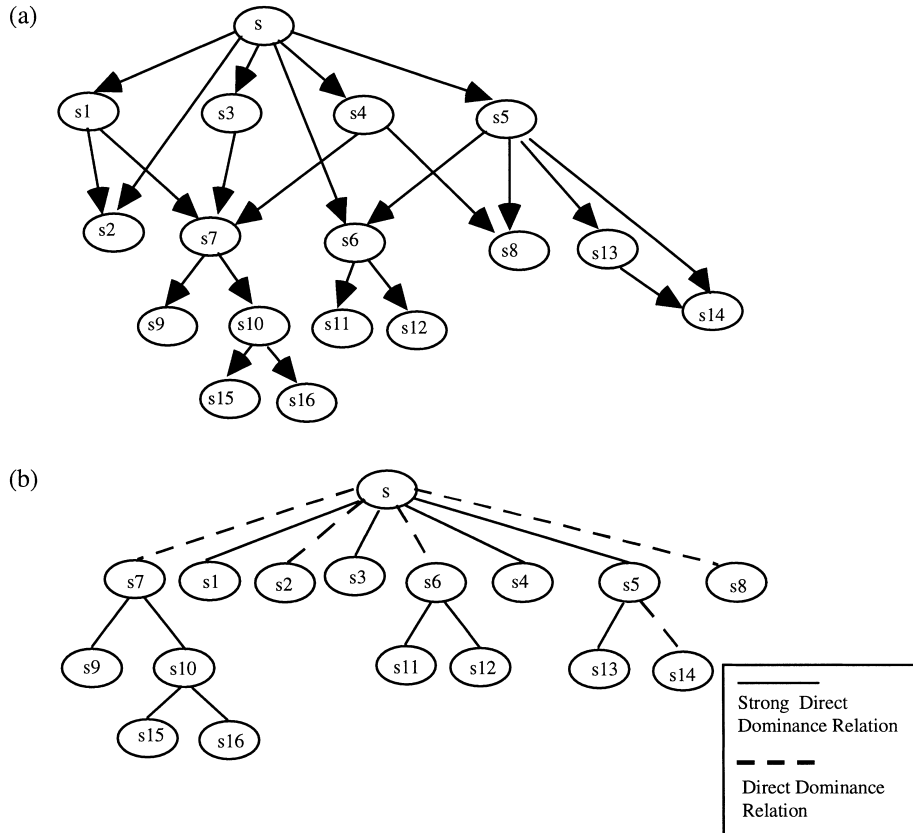


Fig. 2. A call graph (a) and the related call dominance tree (b).

## 6. Experimental results

This section describes the results obtained by applying the method described in the previous sections in two pilot projects. Both projects involved analysis of software systems in current use, written in COBOL, and with all the typical symptoms of ageing legacy systems (lack of documentation, degradation of the structure, technological obsolescence). The main difference was in the size of the systems analysed and the complexity of the host hardware/software platform. In both cases, analysis of the source code was supported by a commercial tool whose scripting language was used to produce reports serving human comprehension activities (such as synonym/homonym analysis) and to create views to be exported to the tools implementing automatic activities, such as association of program units to data stores and dominance analysis.

### 6.1. A case study

The aim of the first project was to assess the proposed method, and identify and understand its strengths and limitations (see (Cimitile et al., 1997) for the details of the experiment). The explorative character of this experiment prompted the choice of a medium-sized soft-

ware system with no particular problems of interface with the host platform. The case study selected is a university hall and residence information system written in Microfocus COBOL at the beginning of the '80s. The system was conceived for a PC environment and uses operating system files to store permanent data; it does not present any TP monitor interface and DBMS transaction management problems.

The system is composed of 103 source programs and 90 copybooks describing file records and screen formats: the overall size of the system is approximately 200 KLOC. The software system was designed using a functional decomposition approach; different functions are implemented in different sub-systems. All programs are well structured in routines implemented as COBOL paragraphs; the verb *perform* is mainly used to activate paragraphs, while *goto* statements are used to reach paragraphs located at the end of some programs (these are clearly the result of maintenance and evolution activities). For programs between 500 and 1000 LOC, the average number of paragraphs is about 31, with an average of 23 LOC per paragraph.

The first step in the method is the identification of the persistent data stores implementing the data structures of the objects. Static analysis of the source code (in particular analysis of the *input-output section* in the

*environment division* and the *file section* in the *data division*) enabled 52 disk files to be identified, reduced to 45 potential objects after solution of synonym/homonym problems. The latter solution was achieved manually; each of the resulting objects was associated with a concept of the application domain. In particular, seven files had simultaneously homonymous and synonymous files: the same physical file was used as a temporary file in different contexts and with a different logical structure, mainly to sort the synonymous files. Each of these was associated in each context to the object identified by the corresponding synonymous file. In other cases, the record structure of the data store was derived from the record structures of synonymous files by joining the descriptions at the maximum level of detail: in this case the synonyms were different views of the same entity of the application domain.

The association of programs to objects based on the minimisation of coupling entails calculating the costs of the accesses made by programs to data stores. To this aim, disk file operations were given different weights depending on their type. The measure of the coupling between a program and a data store was then computed as the weighted sum of the number of accesses the program made to the data store. Different choices of weights were tried and the objects obtained for each of them were analysed and compared against the following criteria: the *identifiability* of the object, that is the meaningfulness of a potential object with respect to the application domain; the *purity* of the object, that is a potential object's capability to include only methods it is responsible for, and the *completeness* of the object, that is a potential object's capability to include all methods it is responsible for.

The best results were achieved using the weight 0.6 for input operations and the weight 1 for output operations. In particular, the weight 0.6 was assigned to *read* and *start* statements, whereas the weight 1 was assigned to *write*, *rewrite*, and *delete* statements; the *sort* statement was associated with the weight 0.6 with reference to the input file and with the weight 1 with reference to the output file. The threshold used to distinguish between predominant and uniform coupling of programs to data stores was 0.6; this was again obtained at the end of the tuning phase. Among the 103 programs composing the system, 63 programs contained accesses to disk files. These were classified as follows:

- 35 programs coupled exclusively to one data store;
- 18 programs coupled predominantly to one data store;
- 10 programs with a uniform distribution of accesses to two or more files (in one case a program was equally coupled to two data stores).

A concept assignment process conducted at the end of this first phase revealed the existence of clusters cor-

responding to meaningful concepts of the application domain; examples are the *college room*, the *system user*, the *room occupancy register*, the *occupant account*, with their corresponding operations. Only 14 potential objects were not associated with any program, although they were accessed by programs characterised by predominant or uniform coupling.

Dominance analysis, normally used to decompose only programs with uniform coupling to more than one data store, was also applied to the 18 programs with predominant access and to three programs with exclusive access to one data store<sup>4</sup>, besides the 10 programs with uniform distribution. This decision is attributable to the explorative character of the case study, whose aim was to understand and characterise the differences in quality of the objects with methods defined at the program level or at the routine level, respectively.

The decomposition of the three programs with exclusive access resulted in the identification of more refined operations: one meaningful example was the program *sysadm* associated as a method of the object *system user* and decomposed into elementary operations, such as *create user*, *get user*, *update user* and *delete user*. The three programs were decomposed into 16 methods. Three programs with uniform accesses could not be decomposed into meaningful operations of different objects, because the routines of these programs uniformly accessed different data stores. Slicing techniques may be used to isolate the different operations in these programs. Significant results were achieved from the decomposition of the remaining 25 programs (18 with predominant accesses and 7 with uniform accesses): 11 programs were completely decomposed into operations of different objects with exclusive access (a total of 66 methods distributed on 18 objects), while the other 14 programs also contained routines with uniform access. However, 145 methods with exclusive access were extracted from these 14 programs and distributed on 16 objects and only 35 routines contained accesses to more than one data store (each of them accessed 2 or at most 3 data stores). Each of the identified object methods was associated with a meaningful function.

In two cases the presence of accesses to different data stores in the same program or routine was used to gain an understanding of the relations between the data stores; on the basis of these relations, it seemed best to group the data stores together into the state of a unique object. The program decomposition process also revealed cases of duplicated code (30 routines reused in different programs) and dead code (40 routines never reached in different programs).

<sup>4</sup> For these three programs, the concept assignment process revealed the possibility of decomposing them into more refined methods, making the corresponding objects more reusable.

## 6.2. Scaling up to large software systems

The aim of the second pilot project was to evaluate whether the method could be re-scaled to complex large systems characteristic of most public organisations. The software system analysed had evolved during the last 20 years to satisfy particular needs of the Health Service in managing pharmaceutical products. The study is still in progress: analysis of the persistent data stores and identification of the potential objects has already been performed, together with the association of methods to data stores at the program level; currently, dominance analysis is being performed to identify methods at the routine level.

The system analysed runs on mainframe, exploits the TP monitor CICS from IBM, two types of DBMS, hierarchical (IMS) and relational (DB2), and a proprietary industrial macro-set. The system is composed of 799 COBOL and 2 ASSEMBLER programs (about 40.5 Mb, 550 KLOC), 260 copybooks (about 1.3 Mb, 16 KLOC), 339 screen maps (about 4.5 Mb, 100 KLOC), 275 files PSB (about 900 Mb, 16 KLOC). The external functions of the system are controlled by 93 JCL jobs.

A preliminary analysis of the source code and the scarce available documentation revealed that the system had historically been developed on the hierarchical database IMS (497 programs contain accesses to IMS segments), which constitutes the only database containing data defining the objects of the application domain. The use of the relational database is marginal (only 74 programs contain embedded SQL code): it was introduced later to import (export) data from (to) databases of external organisations. Indeed, external organisations can provide the data needed by the software system analysed through either VSAM files or DB2 tables. Hence, relational database tables have not been taken into account when identifying the persistent data stores implementing the states of the application domain objects.

Besides importing external data, the system allows input and modification of data in the IMS database through 288 interactive programs containing CICS transactions. However, the most meaningful analyses, such as report generation, are all realised in COBOL by exporting the relevant data from the IMS database to VSAM files. Hence, these files were analysed to identify the data stores corresponding to the application domain objects. Using synonym and homonym analysis, these were then associated to segments of the IMS database. As a result, 294 logical files, 310 physical files, and 76 IMS segments were identified and reduced to 143 data stores after the resolution of synonyms/homonyms and the elimination of printer files. This marked reduction is due to a large number of printer files and synonyms resulting from exporting the same IMS database segments into different files.

Identification of the object methods was conducted using the same weights and the same threshold value as in the first pilot project. The association of interactive programs to the corresponding objects was immediate, as these programs implemented simple data input or updating operations and in most cases had exclusive or predominant accesses to the corresponding data stores. On the other hand, the percentage of programs with exclusive or predominant accesses was very low for programs containing embedded SQL code. This is essentially due to two factors: (i) the models of the data of the two databases (relational and hierarchical) are very different and so there was no correspondence between the relational tables and segments of the hierarchical database; (ii) the same program imports/exports conceptually non homogeneous data. Although dominance analysis has not yet been performed, for programs containing embedded SQL code a good decomposition in paragraphs has been discovered, which enables meaningful operations to be identified at a lower level of granularity.

Analysis at the program level identified about 200 programs predominantly or exclusively coupled to one data store. The next step in this pilot project will be the application of dominance analysis to the approximately 450 programs with uniform access (150 programs have not yet been considered because they do not access neither files nor IMS segments). It is worth to stress that if printer files were included in this analysis, the number of programs predominantly or exclusively coupled is about 350: this is due to the high number of programs printing reports with data from different data stores.

A preliminary analysis conducted at the end of this phase lead to the identification of several programs that could be decomposed into more elementary operations by dominance analysis. A meaningful example is a group of 26 programs all including the same main routine; this main routine invokes different mutually exclusive operations, each implemented by a section, depending on the value of a parameter.

## 7. Conclusion

In this paper a method for carrying out object-oriented decomposition of procedurally oriented legacy systems has been presented. The method focuses on the identification of objects whose data structure is centred around persistent data stores, such as files and tables in the database. The association of methods to objects is driven by the minimisation of object coupling, measured by the number of accesses programs and routines make to persistent data stores. A top-down approach entailing different granularity levels has been followed: programs and routines (or groups of routines) defining object methods are identified in different steps.

The results from two pilot projects have been presented, where the method for identifying objects has been applied to COBOL software systems. The results of these projects support the decision to search for coarse-grained persistent objects. In both case studies the application of the method resulted in the identification of a meaningful number of application domain objects and in a good decomposition of the system that simplified its comprehension. The effort required for applying the method was fairly low, as most of the activities involved were conducted automatically; the human application domain expert was necessary for solving problems of synonyms and homonyms between persistent data stores and in the concept assignment process. For example, the total effort required to conclude the first experiment was 1 man/month. Due to the choice of optimising design metrics, the method produces a good quality system decomposition that can be used as a starting point for a migration process. However, the coarse-grained objects produced are application domain dependent and not easily reusable in other contexts.

The experiments showed that the level of the results obtained depends on the quality of the software system and in particular on its modularisation level. If operations are searched for at the program and routine level, the method might produce scarcely meaningful operations that could be difficult to associate with a particular object, whenever programs and routines had a low cohesion level. This aspect is particularly critical for the COBOL language, where paragraphs or sections can be used depending on the programmer's style and on the coding standards adopted. Whenever programs and routines cannot be associated as object methods, program slicing techniques can be used to obtain a further decomposition level. These techniques have not yet been investigated in the present work. The experiments also revealed cases of highly cohesive routines accessing data stores of different objects that could not be decomposed into methods of different objects: in these cases they could represent relationships between the accessed objects and can suggest a way to identify more complex objects resulting from grouping the features of simpler objects. Some cases of inheritance between objects were also discovered (Cimitile et al., 1997).

Further work is concentrating on the definition of a method based on coupling metrics and on the analysis of the system call graph to take into account programs that do not access persistent data stores. To this end, the system call graph can be modelled by a relation  $CALLS \subseteq PP \times PP \times \mathcal{R}$  and represented by an  $n \times n$  matrix whose rows and columns correspond to programs and whose generic entry  $\gamma_{ij}$  is the weight (e.g., the number) of calls program  $i$  makes to program  $j$ . Hypotheses can be made about the way programs are related. For example, if a program only calls programs that have been assigned to one object, then it can be

considered as a more complex method of such an object. Similarly, if a program is only invoked by methods of one object it can be considered as a private method of this object. Programs that are called by methods of different objects can be clustered into an object implementing general utility services, while a program invoking methods of different objects may be considered as a method of an object obtained by grouping the accessed objects. More precise results can be obtained by modelling this problem by an objective function  $f(X)$  on the variables  $x_{ij}$  based on object coupling minimisation.

Future development of this research activity will be in the direction of studying problems related to the re-engineering of an existing software system, based on the decomposition produced by the method proposed in this paper. Indeed, the successful reuse of objects generated to migrate an existing system to more advanced technological platforms depends on the possibility of building suitable object wrappers at low cost and without excessively degrading the system's performance.

### Acknowledgements

This work was supported by "Progetto Strategico CINI-CNR – Informatica nella Pubblica Amministrazione - Sottoprogetto PROGRESS: Process-Guided REengineering Support System".

### References

- Achee, A.L., Carver, D.L., 1994. A greedy approach to object identification in imperative code. Proceedings of the Third IEEE Workshop on Program Comprehension, Washington, DC, IEEE Computer Soc. Press, Silver Spring, MD, pp. 4–11.
- Belady, L.A., Evangelisti, C.J., 1981. System partitioning and its measures. *Journal of Systems and Software* 2 (1), 23–29.
- Biggerstaff, T.J., Mitbander, B.G., Webster, D.E., 1994. Program understanding and the concept assignment problem. *Communications of the ACM* 37 (5), 72–83.
- Breuer, P.T., Haughton, H., Lano, K., 1993. Reverse-engineering COBOL via formal methods. *Journal of Software Maintenance: Research and Practice* 5, 13–35.
- Calliss, F.W., Cornelius, B.J., 1990. Potpourri module detection. In: Proceedings of IEEE Conference on Software Maintenance, San Diego, CA, IEEE Computer Soc. Press, Silver Spring, MD, pp. 46–51.
- Canfora, G., Cimitile, A., Munro, M., 1993. A reverse engineering method for identifying reusable abstract data types. In: Proceedings of the First IEEE Working Conference on Reverse Engineering, Baltimore, Maryland, IEEE Computer Soc. Press, Silver Sand, MD, pp. 73–82.
- Canfora, G., Cimitile, A., Munro, M., 1996. An improved algorithm for identifying reusable objects in code. *Software Practice and Experiences* 26 (1), 24–48.
- Canfora, G., Cimitile, A., Visaggio, G., 1995. Assessing modularization and code scavenging techniques. *Journal of Software Maintenance: Research and Practice* 7, 317–331.
- Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20 (6), 476–493.

- Choi, S.C., Scacchi, W., 1990. Extracting and restructuring the design of large systems. *IEEE Software* 7 (1), 66–71.
- Cimitile, A., De Lucia, A., Di Lucca, G.A., Fasolino, A.R., 1997. Identifying objects in legacy systems. In: *Proceedings of the Fifth IEEE International Workshop on Program Comprehension*, Dearborn, Michigan, IEEE Computer Soc. Press, Silver Sand, MD, pp. 138–147.
- Cimitile, A., Visaggio, G., 1995. Software salvaging and the call dominance tree. *The Journal of Systems and Software* 28 (2), 117–127.
- De Lucia, A., Di Lucca, G.A., Fasolino, A.R., Guerra, P., Petruzzelli, S., 1997. Migrating legacy systems towards object-oriented platforms. In: *Proceedings of the IEEE International Conference on Software Maintenance*, Bari, Italy, IEEE Computer Soc. Press, Silver Sand, MD, pp. 122–129.
- Dietrich, W., Nackman, I., Gracer, L., 1989. Saving a legacy with objects. In: *Proceedings of OOPSLA*, pp. 77–88.
- Ferrante, J., Ottenstein, K.J., Warren, J., 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9 (3), 319–349.
- Gall, H., Klösch, R., 1995. Finding objects in procedural programs: An alternative approach. In: *Proceedings of the Second IEEE Working Conference on Reverse Engineering*, Toronto, Canada, IEEE Computer Soc. Press, Silver Sand, MD, pp. 208–216.
- George, J., Carter, B.D., 1996. A strategy for mapping from function oriented software models to object oriented software models. *ACM Software Engineering Notes* 21 (2), 56–63.
- Graham, I., 1994. *Migrating to Object Technology*, Addison-Wesley, Reading, MA.
- Hecht, M.S., 1977. *Flow Analysis of Computer Programs*, Elsevier, Amsterdam.
- Horwitz, S., Reps, T., Binkley, D., 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12 (1), 26–60.
- Hutchens, D.H., Basili, V.R., 1985. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering* 11 (8), 749–757.
- Jacobson, I., Lindström, F., 1991. Re-engineering of old systems to an object-oriented architecture. In: *Proceedings of OOPSLA'91*, ACM Sigplan Notices, vol. 26, no. 11, pp. 340–350.
- Lakhotia, A., 1997. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software* 36, 211–231.
- Lindig, C., Snelling, G., 1997. Assessing modular structure of legacy code based on mathematical concept analysis. In: *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, ACM Press, New York, pp. 349–359.
- Liu, S., Wilde, N., 1990. Identifying objects in a conventional procedural language: An example of data design recovery. In: *Proceedings of the IEEE Conference on Software Maintenance*, San Diego, CA, IEEE Computer Soc. Press, Silver Sand, MD, pp. 266–271.
- Livadas, P.E., Johnson, T., 1994. A new approach to finding objects in programs. *Journal of Software Maintenance: Research and Practice* 6, 249–260.
- Müller, H.A., Uhl, J.S., 1990. Composing subsystem structures using (K,2)-partite graphs. In: *Proceedings of the IEEE Conference on Software Maintenance*, San Diego, CA, IEEE Computer Soc. Press, Silver Sand, MD, pp. 12–19.
- Newcomb, P., Kotik, G., 1995. Reengineering procedural into object-oriented systems. In: *Proceedings of the Second IEEE Working Conference on Reverse Engineering*, Toronto, Canada, IEEE Computer Soc. Press, Silver Sand, MD, pp. 237–249.
- Ogando, R.M., Yau, S.S., Liu, S.S., Wilde, N., 1994. An object finder for program structure understanding in software maintenance. *Journal of Software Maintenance: Research and Practice* 6 (5), 261–283.
- Ong, C.L., Tsai, W.T., 1993. Class and object extraction from imperative code. *Journal of Object Oriented Programming*, 58–68.
- Patel, S., Chu, W., Baxter, R., 1992. A measure for composing module cohesion. In: *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, pp. 38–48.
- Reasoning Systems, Inc., *REFINE™ User's Guide*, version 3.0, Palo Alto, CA, 1990.
- Schwanke, R., 1991. An intelligent tool for reengineering software modularity. In: *Proceedings of the 13th International Conference on Software Engineering*, Austin, TX, USA.
- Sneed, H.M., 1993. Migration of procedurally oriented COBOL programs in an object-oriented architecture. In: *Proceedings of the IEEE Conference on Software Maintenance*, Orlando, FL, USA, IEEE Computer Soc. Press, Silver Sand, MD, pp. 396–404.
- Sneed, H.M., 1995. Planning the reengineering of legacy systems. *IEEE Software* 12 (1), 24–34.
- Sneed, H.M., 1996. Encapsulating legacy software for use in client/server systems. *Proceedings of the Third IEEE Working Conference on Reverse Engineering*, Monterey, CA, IEEE Computer Soc. Press, Silver Sand, MD, pp. 104–119.
- Sneed, H.M., 1996. Object-oriented COBOL recycling. In: *Proceedings of the Third IEEE Working Conference on Reverse Engineering*, Monterey, CA, IEEE Computer Soc. Press, Silver Sand, MD, pp. 169–178.
- Siff, M., Reps, T., 1997. Identifying modules via concept analysis. In: *Proceedings of the IEEE International Conference on Software Maintenance*, Bari, Italy, IEEE Computer Soc. Press, Silver Sand, MD, pp. 170–179.
- Subramaniam, G.V., Byrne, E.J., 1996. Deriving an object model from legacy FORTRAN code. In: *Proceedings of the IEEE International Conference on Software Maintenance*, Monterey, CA, IEEE Computer Soc. Press, Silver Sand, MD, pp. 3–12.
- Weiser, M., 1984. Program slicing. *IEEE Transactions on Software Engineering* SE-10 (4), 352–357.
- Wille, R., 1981. Restructuring lattice theory: An approach based on hierarchies of concepts. In: Rival, I. (Ed.), *Ordered Sets*, NATO Advanced Study Institute, pp. 445–470.
- Yeh, A.S., Harris, D.R., Rubenstein, H.B., 1995. Recovering abstract data types and object instances from a conventional procedural language. *Proceedings of the Second IEEE Working Conference on Reverse Engineering*, Toronto, Canada, IEEE Computer Soc. Press, Silver Sand, MD, pp. 227–236.
- Zimmer, J.A., 1990. Restructuring for style. *Software Practice and Experience* 20 (4), 365–389.
- Zuylen, H. (Ed.), 1993. *The REDO Compendium of Reverse-Engineering for Software Maintenance*, Wiley, New York.

**Aniello Cimitile** received the Laurea degree in Electronic Engineering from the University of Naples, Italy, in 1973. He is currently a full Professor of Computer Science at the University of Sannio in Benevento, Italy. Previously, he was with the Department of 'Informatica e Sistemistica' at the University of Naples 'Federico II'. Since 1973 he has been a researcher in the field of software engineering and his list of publications contains more than 100 papers published in journals and conference proceedings. He serves in the program and organising committees of several international conferences and in the editorial and reviewer committees of several international scientific journals in the fields of software engineering and software maintenance. He is a co-editor in chief of the *Journal of Software Maintenance: Research and Practice*. He has been responsible for many international and national applied research projects. His research interests include software maintenance and testing, software quality, reverse engineering, and reuse reengineering.

**Andrea De Lucia** received the Laurea degree in Computer Science from the University of Salerno, Italy, in 1991, the M.Sc. degree in Computer Science from the University of Durham, UK, in 1995, and the Ph.D. degree in Electronic Engineering and Computer Science from the University of Naples "Federico II", Italy, in 1996. He is an assistant professor of Computer Science at the Faculty of Engineering of the

University of Sannio in Benevento, Italy. Previously, he has been with the Department of “Informatica e Applicazioni” of the University of Salerno, Italy, with the Department of “Informatica e Sistemistica” of the University of Naples “Federico II”, Italy, and with the Centre for Software Maintenance of the University of Durham, UK. He serves in the program and organising committees of conferences in the field of software maintenance and program comprehension. His research interests include reverse engineering, reuse, reengineering, migration, program comprehension, and visual languages.

**Giuseppe A. Di Lucca** received the Laurea degree in Electronic Engineering from the University of Naples “Federico II”, Italy, in 1987 and the Ph.D. degree in Electronic Engineering and Computer Science from the same University in 1992. He is currently an assistant professor of

Computer Science at the Department of “Informatica e Sistemistica” of the University of Naples “Federico II”. His research interests include software engineering, software maintenance, reverse engineering, software reuse, software reengineering, and software migration.

**Anna Rita Fasolino** received the Laurea degree in Electronic Engineering (cum laude) from the University “Federico II” of Naples, Italy, in 1992, and the Ph.D. in Electronic Engineering and Computer Science from the same University in 1996. From 1992 to 1995 she was a Ph.D. student at the Department of “Informatica e Sistemistica” of the University of Naples “Federico II”. She is currently an assistant professor of Computer Science at the University of Bari, Italy. Her research interests include software maintenance, software quality, reuse, reverse engineering, and software migration.