A SURVEY OF OBJECT IDENTIFICATION IN SOFTWARE RE-ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1998-6

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1998-6, APRIL 1998

A SURVEY OF OBJECT IDENTIFICATION IN SOFTWARE RE-ENGINEERING

MAARIT HARSU

University of Tampere
Department of Computer Science
P.O.Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-4352-7
ISSN 0783-6910

A survey of object identification in software re-engineering

Maarit Harsu

Department of Computer Science
University of Tampere
P.O. Box 607, FIN-33101 Tampere, Finland
e-mail: `csnima@cs.uta.fi`

Abstract

In order to translate a non-object-oriented (procedural) program into an object-oriented one, objects must be identified from the procedural program. Object-oriented programs (compared with procedural ones) are considered to be easier to reuse and maintain. Thus, object identification followed by translation from a non-object-oriented language into an object-oriented language is one way to re-engineer legacy programs. This paper gives an overview of re-engineering in general and of object identification especially. Associated with object-orientation, identification of (design) patterns is discussed, too.

Keywords: Re-engineering, object identification, re-engineering patterns, software maintenance.

1 Introduction

When re-engineering legacy programs, old programs are modified into a new form to make them more compact and structural, and easier to understand, maintain, and reuse. This is a way to improve software quality. An important point in re-engineering is to find those software parts which can be reused. In translation into an object-oriented language, it is important to find reusable components and objects from the procedural programs. The easiest way to start to use an object-oriented language is to wrap the old code into a black-box module with an object-oriented interface, as has been done in [18]. The functionality of the program is preserved as such. However, the old part of the program must be reused as whole, the components of the program are impossible to be reused separately. In this paper, we are trying to provide better ways to reuse legacy programs.

An important subarea in software re-engineering is modifying old programs into object-oriented platform. Advantages of object-oriented programs are considered to be encapsulation, data abstraction, information hiding, etc. In addition, object-oriented programs are easy to maintain and reuse. The purpose of this paper is to introduce known means to translate procedural programs into object-oriented ones.

When translating non-object-oriented programs into object-oriented programs, direct source-to-source conversion is not possible [25]. Instead, the translation requires more abstract view about the source program which can be achieved by the means of re-engineering. During the reverse engineering phase, the objects are identified from the procedural code. Then the actual translation can be performed according to the source code and to the identified and accepted objects.

This paper proceeds as follows. The second section introduces basic concepts about re-engineering. Section 3 considers re-engineering in general. Section 4 discusses a subarea of re-engineering, namely identifying objects from code in order to convert programs into an object-oriented language. That section introduces some possible ways to object recovery. Section 5 considers re-engineering especially from the point of view of (design) patterns. In section 6, the problems concerning re-engineering and object identification are discussed. Finally, in section 7, we draw conclusions.

2 Terminology of re-engineering

In this section, we introduce and define some basic concepts concerning re-engineering. This section, excluding the last term, is based on [11]. For simplicity, the definitions assume that the software life-cycle consists of three phases: requirements analysis, design, and implementation.

Software maintenance
The ANSI definition of software maintenance is the "modification of a software product after delivery to correct faults, to improve performance or other

attributes, or to adapt the product to a changed environment". The first step in software maintenance is to examine the program to understand it. Reverse engineering facilities can be used to support the maintenance process. Thus, reverse engineering is the part of the maintenance process helping to understand the program in order to make the desired changes. Maintenance can also considered as reuse-oriented software development [3].

Forward engineering

Forward engineering is the traditional process of moving from the requirements of the system to its design, and from design to the concrete implementation of the system. Actually, forward engineering means exactly the same as engineering. The adjective forward is used to distinguish the term from reverse engineering.

Reverse engineering

Reverse engineering is a reverse process for forward engineering. In reverse engineering, the extracted information about a system is more abstract than the system itself under examination. For example, abstractions or design decisions are generated from the implementation level. Reverse engineering can start from any level of abstraction or at any stage of the life-cycle. Reverse engineering does not involve changing the subject system. It is a process of examination, not of change or of replication.

Reverse engineering has many subareas, the most important ones of which are redocumentation and design recovery. Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternative views (e.g. data flow, data structure, control flow), which help human understanding the system better. The aim of redocumentation is to recover documentation that existed or should have existed. Examples of redocumentation tools are pretty printers, diagram generators, and cross-reference listing generators. The aim of these tools is to provide easier ways to visualize relationships between program components.

Another important subarea of reverse engineering is design recovery. In design recovery, the needed information is collected besides by examining the system itself, also from domain knowledge and by using other existing (design) documents. Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software engineering representations or code.

Restructuring

Restructuring is the transformation from one representation form to another at the same abstraction level. The transformation preserves the external behavior of the system. Restructuring is typically used in implementation stage to transform code from an unstructured form to a structured form (transforming, for example, goto-statements to control structures). In addition, restructuring can be used in other stages, too: for example to reshape design plans or requirement structures.

Re-engineering

Re-engineering can also be called both renovation and reclamation. Re-engineering is the examination and alteration of a system to reconstitute it in a new form and the subsequent implementation of the new form. Thus, re-engineering generally includes reverse engineering (to achieve a more abstract description) followed by forward engineering or restructuring. This may include modifications with respect to new requirements not met by the original system.

Reuse re-engineering

Reuse re-engineering can be considered as a subarea of re-engineering, although it is not defined in [11]. The term reuse re-engineering is used in [6, 7, 8, 9], and quite a similar term re-engineering for reuse in [30]. In reuse re-engineering reusable features are identified from the code, especially in the reverse engineering phase. Thus, because objects can be considered reusable features, object identification is a kind of reuse re-engineering.

3 General issues about re-engineering

This section considers some general issues about re-engineering. The relationship between re-engineering and maintenance is discussed first, and after that some re-engineering methods and tools are introduced.

3.1 Re-engineering and maintenance

With re-engineering the software quality can be improved, and the subsequent software maintenance tasks can be performed more easily. Particularly, if reusable components are identified, and the software is re-engineered into a form that better supports reusability and maintainability, the quality of re-engineered software is considerably better than that of the original software.

Software re-engineering is closely related with software maintenance. Swanson has presented three reasons for software maintenance [48]. First, the errors in specification, design and implementation must be corrected (corrective maintenance). Second, the data and processing environments may change (adaptive maintenance). Third, the performance of the software must be maintained (perfective maintenance). As mentioned in chapter 2, also the ANSI definition for software maintenance covers these three aspects. Besides the three reasons, the fourth reason has been recognized later: the requirements of the software product may evolve (preventive or evolution maintenance) [2, 4, 43]. Preventive maintenance is closely connected with re-engineering and reverse engineering. In addition, each of the other types of maintenance can be aided with re-engineering means.

Software under maintenance is usually lacking of important documents or these documents are not updated. Requirements, design documents, maintenance information, etc. can be in the heads of the programmers, but these programmers are not necessarily available. Re-engineering can be used in this kind of information recovery to produce structure charts, data flow diagrams, entity-relationship diagrams, etc.

However, re-engineering is not always the best choice. Figure 1 shows in which situations it is reasonable to re-engineer software. (Figure 1 is from [29].)

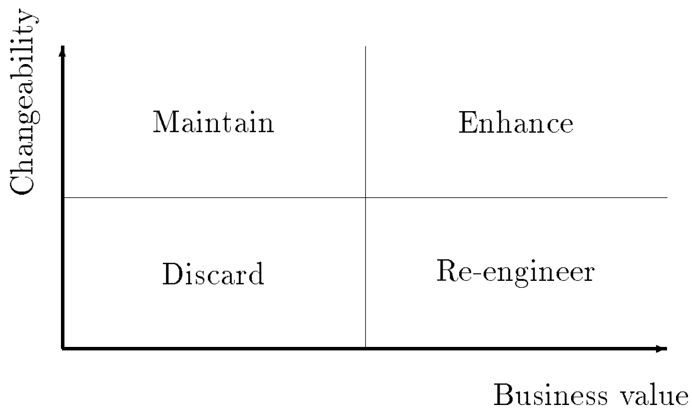

Figure 1. Decision matrix.

If the legacy system is easy to change, it is reasonable to just maintain it. If the system is difficult to change, but it has a high business value, more radical changes are needed and the system is worthwhile to be re-engineered. A system with a high changeability or a low business value either does not need re-engineering or is not worth of it.

3.2 Basic methods used in re-engineering

There are many ways and methods to re-engineer a software system. Dependency analysis shows the dependencies between language structures (modules, data objects, functions). The simplest method of this kind is the cross-reference listing available in many compilers. This listing typically shows where different identifiers are referred, and thus, reduces the effort in searching through program listings. Another simple way to get dependency information is to construct a calling dependency graph, where a procedure is connected to another procedure if one of the procedures calls the other.

Common methods in re-engineering are those providing flow graphs. Flow graphs are typically distinguished into control flow and data flow graphs. To construct a control flow graph, the statements of a program are organized into basic blocks. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The basic blocks are nodes of the control flow graph. The edges represent transfers of control between the basic blocks. For example, an if-statement is presented as branching from the condition node to two nodes, one being the basic block of the true branch and the other being the basic block of the false branch. For another example, a while-statement forms a cycle in the graph. The end of the while-statement is directly connected to the condition of the while-statement. The opposite connection flows via the basic blocks of the body of the while-statement. Data flow dependencies occur between data objects when the value held by one object may be used to calculate or set by another object. Flow graphs are considered for example in [1, 50]. Originally, flow graphs are constructed for compilers and particularly for optimizing code. However, they can be useful in re-engineering, too.

Program slicing, first introduced by Weiser [49], can be used to aid re-engineering. Program slicing is a decomposition based on data flow and control flow analysis. Starting from a subset of a program's behavior, slicing reduces the program to a minimal form which still produces that behavior. For example, if we slice a program according to a certain variable, the program slice contains those program lines that refer to the variable (definitions and usages) and some other necessary lines. For example, if the variable is referenced inside an if-statement, the head and the end of the if-statement must be included in the slice.

3.3 Tools used in re-engineering

Many tools supporting re-engineering have been introduced in literature. Some of the tools are based on lexical structure of the program, and some others require parsing the program. The tools based on lexical structure are usually simple and easy to implement. However, finding some structures may need parsing the program. Parsing-based tools have their deficiencies, too [5]. Although they suit very well to the recognition of programming-oriented concepts, they cannot be used in searching for semantic information.

Some tools search for certain concepts or patterns from a program. UNIX provides grep command for searching for certain strings and regular expressions. However, in re-engineering this is not always sufficient. The tool described in [41] is based on pattern matching, which is implemented using syntax-directed approach. The following searches are possible:

- Find all while-statements where the condition of the while-statement is a relational expression of the form not-equal-to zero.
- Find all occurrences of three consecutive if-statements.
- Find situations where the values of two variables are being swapped.
- Find a structure of three nested loops.

These kinds of things are hard to find with conventional tools, like grep command.

There are a lot of redocumentation tools that show useful information about the program. This information can be called source model [35], and it consists of the information described in the previous subsection such as data and control flow graphs, call graphs, etc. There are several papers about these topics. Cimitile and De Carlini represent algorithms for generating source models [12]. Their algorithms use an algebraic representation of program modules. Jackson and Rollins introduce a new form of the program dependence graph which is in certain situations more suitable than the original program dependence graph [28]. Murphy and Notkin introduce a system extracting different source models from lexical specifications [35]. Cleveland describes a tool providing multiple views about a program and the possibility to move between the views [13]. One simple and quite a useful tool is such that analyzes differences between two versions of a program in order to understand the change made in one of program versions [50].

An interesting approach is described in [10]. The authors suggest a parsing-based system which collects information about the relationships between language structures. It saves information about object domains, such as files, macros, data types, global variables and functions. Each object domain has attributes. For example a function has the following attributes: the name of the file containing it, its (return) type, its name, static variables used in it, its beginning and ending lines. The relationships between each object domain are stored in the database. Thus, the normal database queries can be made. For example, the following queries are possible:

- Which functions defined in a certain file are referred to by the functions in another file?
- Which functions refer to a certain global variable and a certain data type?
- Which static functions refer to a certain macro?

This system provides also graphical views about the relationships.

There are tools for visualization of programs under re-engineering and specialized editors for maintenance and re-engineering. Rajlich et al. introduce an editor for Fortran programs [45]. With this tool, programs can be edited both as text and as a graph. Quite a similar tool for C programs is introduced in [32]. Ning et al. introduce an editor for Cobol programs [38, 39]. The authors define program segments to consist of statements which are semantically related, but not necessarily physically adjacent. With the tool, these kinds of program segments can be isolated from other program text to examine them more precisely. In addition, program slicing is possible.

Cordy et al. have a different approach [17]. They introduce a tool whose comprehensive viewing paradigm is borrowed from program development environments. This paradigm is source text elision. By using the editor, meaningless parts of the program can be elided. The body of a procedure can be elided, only showing the header of the procedure. Similarly, the body of a control structure can be elided, only showing the condition and the ending of the control structure. This kind of tool makes program browsing easier. In other context than re-engineering, quite similar tools based on hypertext [19] and on active text [36] are also introduced.

As shown there are many kinds of tools supporting software maintenance and re-engineering. The tools introduced in this subsection typically show the program points requiring modification or help in finding them. The actual modification is then performed by a human.

4 Searching for objects

This section describes several different systems searching for objects in conventional (procedural) programs, and thus, enabling translation from a procedural language into an object-oriented one.

4.1 Gradual objectifying

Jacobson and Lindstöm show how old systems can be gradually re-engineered to an object-oriented architecture [29]. They consider different situations in re-engineering depending on whether the change concerns implementation technique or functionality. They discuss three different situations: a complete change of implementation technique and no change in the functionality, a partial change in implementation technique and no change in functionality, and a change in functionality.

Consider the situation when the implementation technique changes completely, but the functionality does not change at all. The authors use any available documents of the old system: requirements specifications, user operating instructions, maintenance manuals, training manuals, design documentation, source code files, database schema descriptions, etc. From these elements they prepare an analysis model by using object-oriented analyzing methods, e.g. [15, 14]. Then the authors map each object obtained from analysis model to the implementation of the old system. They redesign the system using a forward engineering technique for object-oriented system development.

If the implementation technique changes only partially, the new part (object-oriented) and the remaining part of the old system (non-object-oriented) must be made fit together. Some interface is needed between these different parts via which they can communicate with each other.

The third situation is such that only the functionality changes. The analysis model is changed according to the requirements on changes in functionality. Forward engineering is performed to obtain new objects. Objects identified from the old system either are deleted or receive new attributes.

The authors only present general rules to re-engineer a system to an object-oriented architecture. They do not precisely tell how to modify existing code to object-oriented code.

4.2 Object Finder

Object Finder is described in [33, 40]. The authors introduce two methods for finding objects: globals-based object identification and types-based object identification. The former method acts as follows. For each global variable of the program, the set of routines that directly use the global variable is determined. After that a graph is constructed. The nodes of the graph are the routines found in the first step, and the edges connect those nodes (routines) which have common references to the same global variables. Each strongly connected component of the graph defines an object and its routines.

However, the program code may be so tightly coupled that the whole program is one large strongly connected component, and the method cannot split the program into smaller parts. For these situations, the authors propose the following solution. The user specifies (by hand) those global variables which form undesired links between nodes (routines) in the graph. These global variables are then ignored during the globals-based object identification analysis.

Another method for identifying objects is based on types. First, a topological ordering of all types in the program is defined as follows. If type x is used to define type y, then we say that x is a sub-type of y, and y is a super-type of x. The relation of types is transitive: if x is a sub-type of y and y is a sub-type of z, then x is a sub-type of z. Second, routines and types are connected together as follows. A type is connected to a routine, if it is a formal parameter or a return type of the routine. However, if a type and its super-type would be connected to the same routine, only the super-type will be connected. The groups consisting of together connected routines and types form objects and their routines. Again, human assistance is needed to break too large objects.

The reason why super-types are chosen primarily as described above is to eliminate some irrelevant connections. We may, for example, have type *node*, and another type *stack* whose items are nodes. In this case, type *node* is a sub-type of type *stack*. Consider routine *push* having two arguments: one of type *node* (which will be pushed) and another of type *stack* (to which will be pushed). For a human, it is clear that routine *push* should be a routine of *stack*, not a routine of *node*. If we follow the above rules, the connection between routine *push* and type *node* will be removed, and the remaining connection will be the one between routine *push* and type *stack*, as we intended.

4.3 Extensions of Object Finder

Livadas and Johnson [34] use the two methods for identifying objects (globals-based and types-based) described in the previous subsection. However, they have extended these methods as follows. First, in programming languages allowing nested procedures (like Pascal), those variables which are visible for several procedures should be considered as global variables. Second, the following situation may occur. A variable is global for a function and the function passes the variable to another function as a parameter. These connections can be identified with the method proposed by Livadas and Johnson, because they use a graph as an underlying structure representing bindings between the language structures. Third, the authors have noticed that the return type of a procedure (as suggested in the previous subsection) is not always appropriate criteria for forming connections. For example, routine *is_empty* (for a stack) has boolean as its return type. However, the routine is not a routine of type boolean, but a routine of stack.

Besides the two methods (globals-based and types-based), the authors use another method, called receiver-based object identification. Receiver parameter type of a routine means such a type, the parameter of which is modified in at least one execution path of the routine. If a routine has several parameters one of which is modified, the type of the one is typically an object candidate. For example, consider routine *push* having two arguments: one of type *node* and another of type *stack*. The routine pushes the given node to the given stack. Thus, only the parameter of type *stack* is modified, and stack should be an object candidate having the routine *push*.

The objects found by using the above three object identification methods are called primary objects. They are found via queries on the internal program representation. Another set of objects can be found by queries on the primary objects and by some additional information. These objects are called secondary objects. The operations for finding secondary objects are: selection, union, intersection, subtraction, and deletion. In the selection operation, object finding is performed on a subset of routines, types, and global variables. Selection is made according to some property, for example, selecting all routines that access a certain global variable. With the union operation, routines can be grouped based on mixed criteria (the union of several criteria). In intersection, for example, routines having a certain receiver type, but which also access a certain global variable, can be grouped together. In subtraction, routines having a certain property can be excluded from the group. In deletion, some spurious dependencies can be deleted.

In identifying objects, the internal program representation is the system dependence graph. It is a directed, labeled multigraph consisting of a program dependence graph and a collection of procedure dependence graphs. Program dependence graph represents the main program, and procedure dependence graphs represent procedures. Each node of these graphs represents a program construct such as declaration, assignment, etc. Additional nodes represent for example formal and actual parameters. The edges represent several kinds of dependencies among the nodes and are distinguishable according to the labels attached to them. When using system dependence graph, all the information needed can be obtained from the headings of the procedures. Thus, also procedures with lacking bodies (for example library procedures) can be modeled with the system dependence graph.

4.4 Identifying reusable abstract data types

There are several papers about the RE^2 project [6, 7, 8, 9]. The goal of the project is to understand how reverse engineering and re-engineering can promote software reuse. Reusable software components (for example objects) are extracted from existing systems and they are collected into repositories.

According to the authors, searching for objects in code can be divided into five phases:

1. candidature,

2. election,
3. qualification,
4. classification and storage,
5. search and display.

Candidature phase analyzes the source code and chooses some software components as candidates of reusable modules. Election phase analyzes the chosen software components and produces from them reusable modules. Qualification phase produces the specifications of each reusable module obtained in the election phase. Both the functional and the interface specifications are produced in that phase. Classification and storage phase classifies the reusable modules and related specifications according to a reference taxonomy. The aim is to define a repository system containing the reusable modules produced. Search and display phase sets a front end user interface to interact with the repository system. The aim is to make finding the desired modules as simple as possible, for example, by giving visual support to navigate through the repository system. Only the first three phases are discussed in the papers concerning RE^2 project.

The authors use the globals-based and types-based object identification methods introduced with the Object Finder [33, 40] (described in subsection 4.2). However, the authors of RE^2 suggest some improvements for those methods. They have noticed some situations in which the graph-based object identification methods do not suit very well. For example, consider two abstract types: a stack and a queue. A program may have a common initialization routine for both of the types. Or a program may have a routine which copies the elements of the stack to the queue or vice versa. These routines form undesired links between the abstract data types and among the corresponding subgraphs. The former kind of link (e.g. with the common initialization routine) is called coincidental connection and the latter kind of link (e.g. with the copy routine) is called spurious connection. The authors suggest that the routines producing coincidental connections are sliced into several routines via program slicing, and the routines producing spurious connections are just removed.

To find isolated subgraphs from the graph having coincidental and spurious connections, the authors suggest the following solution. They calculate the internal connectivity index of each subgraph. It is the ratio between the number of edges linking couples of nodes in the subgraph and the total number of edges of the subgraph (both internal edges inside the subgraph and edges which connect a node of the subgraph with some external node). This value is always between 0 and 1, and it is 1 for a totally isolated subgraph. The value of internal connectivity changes when routines are merged, sliced, and removed. The whole algorithm for finding objects is iterative, and it stops when the user is satisfied with the internal connectivity index of each resulting subgraph.

4.5 COREM

There are several articles about the COREM system [20, 21, 22, 23, 24, 25, 26, 31]. The authors do not only consider data stores as object candidates, but also examine functional relationships between data structures to get more object candidates from which several are selected to become objects. The authors divide object recovery into four phases: design recovery, application modeling, object mapping, and source-code adaptation.

The first step, design recovery generates different low-level design documents (i.e. structure charts, data flow diagrams). These documents are again modified to generate entity-relationship diagram (ERD). The ERD consists of entities based on the data stores of the data flow diagram. Data structures which are functionally related to these data stores are also added as entities of the ERD. A functional relationship is defined as a manipulation or use of one or several attributes of an entity (e.g. data store) by another entity (e.g. data structure). The relations of the diagram are both special relations (e.g. is-a, part-of) and general relations. The general relations are derived from those procedures of the program that incorporate a functional relation between two entities.

The entity-relationship diagram is transformed into an object-oriented application model, called RooAM (reversely generated object-oriented application model). The transformation is based on the structural similarities of these two design representations: each entity is mapped to an RooAM-object.

The relationships (is-a and part-of) between objects are derived directly from the corresponding structures of the entity-relationship diagram. These relationships can also be directly derived from the declaration parts of the source code: is-a relations can be derived from variant records, while part-of relations can be derived from array or pointer structures within data-type definitions. The operations for an object can be found both from entity-relationship diagram and by examination of the source code to recognize the procedures manipulating the object.

The second phase, application modeling constructs a forwardly generated object-oriented application model (FooAM). This model is independent of the actual procedural implementation, but is based on the same requirements as the examined program. This step needs a human expert who is either experienced in the application domain or who participated in the development of the program under consideration. This modeling can be done by applying any object-oriented analysis methods, for example [15, 14, 46]. Actually, this step models the application, which is earlier modeled by the means of procedural program design, again by using object-oriented techniques.

The third phase, object mapping maps the objects of the two models (RooAM and FooAM) together. The aim of this step is to find a mapping of similarities between the elements of those two models. Because of the different origins, the models have some differences. The FooAM originates from the requirements analysis and the domain knowledge, and therefore does not have a high degree of details (e.g. attributes have no types). The RooAM on the other hand offers a lot of detailed information because it originates from the examined source code. Thus, in mapping objects, the FooAM is working as a pattern for the target application model to which the RooAM gives the detailed information. The two models have varying amount of attributes, instance connections, and message connections. Thus, some parts remain in one of the application models without correspondence. These parts contain elements of the procedural program that cannot be mapped to the target object-oriented model, and they form so called procedural remainder.

In the fourth step, source-code adaptation, both the earlier mapped elements and the procedural remainder are adapted to object-oriented concepts. Syntactically, objects are formed by encapsulating the attributes and the declaration parts of the procedures. Global data items are encapsulated in separate objects, too. They are called data objects, because they have only attributes, but no procedures. The procedural remainder produces the following objects. First, it gives master objects which include the highest level of system control and can be compared to a control unit. A typical example is the main program. Second, it gives aggregation and coordination objects which perform system control over a specified set of procedures and provide some kind of functionality for a master object. Third, it gives provider objects which perform the main functionality (e.g. sorting, searching, computing). These kinds of objects do not perform any system control, but provide a certain functionality for aggregation and coordination objects.

In each phase, a human expert is needed. In the first phase, various ambiguities may arise, for example, to which particular object a procedure should be assigned. These situations can be solved by a human expert. The second phase is implemented totally by a human expert. In the third phase, using application domain knowledge, the human expert can perform the matching of each ambiguous RooAM service candidate to a particular FooAM service. Many parts of the fourth phase can be automated, however, the human expert has to deal with adapting the interfaces of procedures and in the decomposition of the procedural remainder. Especially, the reference [31] considers the means to decrease human intervention. Section 5 considers the COREM system from the point of view of re-engineering patterns.

4.6 Programming plans

Quilici [44] presents how to automatically extract object-oriented design knowledge to support translation from a procedural language into an object-oriented one. He is especially interested in recognition common objects and operations from the old code and replacing them with libraries containing human-generated object-oriented code. He has the following approach to identify objects in the code. He has studied how programmers understand short test programs having uninformative variable names. Then he tries to build a program that mimics their understanding process.

He has noticed that programmers do not try to fit a program fragment under consideration to every programming plan. Instead, programming plans form a hierarchy from general plans to more specialized plans. In addition, some plans can be attached to other plans. For example, if a plan computing distances is found from the program code, it can be assumed that there are also points, the distances of which are to be computed. He has organized a plan library due to these notifications. The plans in the library have links to other plans according to specialization and implication.

4.7 Function abstraction

Philip and Ramsundar [42] consider re-engineering from a procedural language into an object-oriented one. To re-engineer software the authors use function abstractions. They operate the system, invoke all its functions, and evaluate the behavior of the functions. Based on the functionality, they get abstractions for each function. The authors collect each extracted function to the table called function matrix. Each item of the function matrix is associated with a reuse factor, telling the potential for reusing that function in the target system.

After identifying the function abstractions, the authors analyze the corresponding code to refine the functions to lower levels. During this analysis, some data elements are also identified, and they are added to a data dictionary. In addition, the identified blocks of code with specific functionalities are converted to high level algorithms.

To identify objects, the data dictionary is searched to identify items which correspond to specific components in the system. In most cases these items become the objects in the target system. In addition, closely related items in the data dictionary are used as attributes of the found objects. After the objects are identified, they are compared against the remaining data items in the dictionary. Those items which describe an identified object are placed as attributes of that object.

Each element of the function matrix has a corresponding entry in a separate table. This table contains the algorithms of each function. The algorithms refer to the data elements of the system. This information which relates functions to data elements is used to assign functions as operations of the identified objects. Each function is assigned to an object which closely matches the object or the attributes of the object. However, the decision to assign an operation to an object is usually subjective. The relations between two objects can also identified through existing procedure invocations within the algorithms or code. This provides a message path between the objects where the corresponding operations are located.

In addition to the described techniques, usual object-oriented development techniques (for example [46]) are used during the forward engineering phase.

4.8 Automated object identification

Newcomb [37] describes a highly automated process to re-engineer procedural programs into object-oriented ones. His object-oriented model is based on state machines, and it is called hierarchical object-oriented state-machine model. The paper describes the mapping operations and analyses that achieve the resultant abstract structures, but not the final process for translating into a specific target object-oriented language.

The first part of the transformation includes parsing the procedural program to create an abstract syntax tree (AST). The abstract syntax tree is then decorated with semantic properties of the program. The resultant abstract syntax tree is called an augmented abstract syntax tree.

Different kinds of analyses are performed. Scope analysis attaches the declaration and the occurrence of a variable with each other. Scoping rules depend on the semantics of the programming language. Block structured languages usually provide both locally and globally scoped variables. Set-use analysis identifies the usages and definitions of a variable taking into advance the meaning of the usage. Program unit analysis describes the local properties of each data unit and function unit. Examples of local properties are type, the location of definitions and references. Alias analysis examines records and their fields to find out the occurrences of records having a different name but an identical structure. The template for a record having different occurrences is called collision former. The records matching the collision former are considered aliases. An alias map is a relation whose domain is a collision former and whose range is the set of alias records.

For procedures, control flow graphs, a data flow graph, and a state transition graph are constructed. The state transition graph consists of a start state, an end state and a set of intermediate states joined by state transitions. A state transition is defined by each distinct sequence of control conditions followed by a sequence of actions. The state transition table depicts one or more states and the conditions and actions involved in transitions between states.

Different kinds of classes can be derived as follows. Data object classes can be obtained by the alias analysis. The collision former corresponds the object class, and the instances of the class are the alias records. A program object class is a class whose instances are top level programs. There are several classes for different methods: for data transforming methods, for side effecting methods, for primitive methods, and for composite methods.

The object oriented model must be normalized. This involves transformations reducing the complexity of the model by merging similar or identical components into functional or structural unifying generalizations. For data structures, the generalization is the subclass relationship and the instance-class relationship. For functional structures, the generalization is parameterization and method inheritance.

4.9 Extracting object-oriented specification

Sneed and Nyáry present how object-oriented specification can be derived from existing procedural programs and databases [47]. They extract structures, interfaces and algorithms from legacy code and create a program design specification. The program is then reimplemented according to the specification. The functionality of the program is preserved without the constraints of the legacy design. However, the code has to be rewritten and the test data regenerated.

The authors use different sources from which they identify different objects. User interface objects can be derived from panels for communicating with the user. Data objects can be obtained from databases for storing persistent data. File objects comes from job control procedures which refer to data sets. Record objects, view objects, work objects, and link objects are derived from programs.

To extract operations for objects the authors examine cross references. First, they segment the procedures of the program into elementary operations. Then they check which object each elementary operation refers to. After that, the connections between objects must be depicted. Objects connect to each other via parameters and interfaces. Thus, in this step, the parameters of the operations are defined.

The final step is to capture and document the sequence in which the operations are executed. The sequence is determined by separating the program control flow from the program operations. Each operation knows its successor operation, and when an operation has been terminated, the operation sets a certain global variable to point to the next operation. Thus, after the invocation of each operation, the global variable tells the operation which should be invoked next.

5 Re-engineering patterns

There are several kinds of patterns. Probably the most famous patterns are design patterns, introduced by Gamma et al. [27]. Similar kinds of patterns are also introduced by Coad [14]. There are also code patterns which are programming language oriented abstractions of style guidelines [16]. From the point of view of this paper, especially interesting patterns are re-engineering patterns, discussed only in [26]. (In [26] re-engineering patterns are called object patterns or application patterns.)

There is a difference between re-engineering patterns and the other patterns. The other patterns are ideal solutions, guidelines to follow. Instead, re-engineering patterns are code fragments found in programs that have been run and maintained for many years. Thus, they are not patterns to be followed, but patterns to be untangled.

The COREM system described in [26] is discussed in subsection 4.5. In that subsection, we noted that the system has two approaches: forward engineering from the requirements of the software and reverse engineering according to the same software. Similarly, pattern recognition has two approaches, too: pattern-driven scavenging for object patterns in source code and source code -driven capturing of object patterns.

Pattern-driven scavenging uses the reverse engineering approach. Instances of object patterns are tried to find in the source code. However, one cannot expect to find a direct instantiation of a particular object pattern in the source code. Nevertheless, typically a programmer solves similar problems in similar ways. Consequently, identifying the particular solution patterns used by a programmer to implement certain aspects which could be grouped into an object or an abstract data type aids in identifying constituent parts for object patterns.

Source code -driven capturing uses the forward engineering approach. Abstract data types of the procedural program are considered, and they are tried to map to some object patterns. Mapping is performed in several steps, but an exact mapping between a data type and an object pattern is not necessarily found. In addition, for each kind of object pattern the source code -driven capturing is influenced by different constraints. Again, there are some difficulties, because the original problem is not modeled with object-oriented concepts. Some relationships between entities may not be exactly the same relationships in the code. The original programmer may have not seen the relationships clearly enough. For example, she may have not seen the similarities between some concepts, and thus, given them quite independent implementations. Taking advantage of this historical pattern may help in finding the real relationships.

Together these two approaches, pattern-driven scavenging and source code -driven capturing, help finding re-engineering patterns. Note that the method is not totally automated, instead human assistance plays a significant role.

6 Problems in re-engineering and object identification

As introduced in this paper, there are a lot of tools to aid re-engineering and object identification. However, usually the existing tools are not suitable for a certain situation. For example, they do not support the particular dialect of the programming language. Parsing-based tools may not accept a deficient program, for example, the program cannot be parsed if some included files are missing. In addition, usually the tools are not flexible nor extendible.

A typical situation when re-engineering a legacy program is that the only existing document about the program is the code. The other possible documents may be either missing or out of date. The other information about the program is in the heads of the designers and programmers. However, they may have left the company or moved to other duties. The lacking documents make re-engineering more difficult. Thus, many tools for re-engineering do not require any other document than code.

In re-engineering and object identification, domain knowledge is usually needed. Thus, totally automated tools do not yield good resulting code. The tools are typically semi-automatic: routine work is performed automatically, but human assistance is needed in some decisions, for example in the decision whether to accept an object candidate as a final object. However, producing pure automated tools for as many re-engineering phases as possible is desirable, because they can significantly reduce resources needed for re-engineering and object identification.

Object identification area especially has difficulties in recovering objects in non-object-oriented code. When using the graph-based solutions as described in subsections 4.2, 4.3, and 4.4, searching for isolated subgraphs produces satisfactory results only for the ideal programs which have been designed according to a fully object based approach [8]. For other programs the subgraphs are not totally isolated, and human assistance is needed to dissolve the insignificant relationships.

7 Conclusions

In this paper, re-engineering in general and object identification especially have been considered. Several tools for aiding these processes have been introduced in literature. However, most of the tools are not fully automatic, instead human assistance is needed.

Re-engineering and object identification have been investigated quite a lot. However, as stated in the previous chapter considering the problems of these areas, a lot of questions are still lacking of answers, and further research is needed.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] S. Ajila, Software maintenance: an approach to impact analysis of objects change, *Software - Practice and Experience*, **25** (10), 1995, 1155-1181.

[3] V. R. Basili, Viewing maintenance as reuse-oriented software development, *IEEE Software*, **7** (1), 1990, 19-25.

[4] S. Bendifallah, W. Scacchi, Understanding software maintenance work, *IEEE Transactions on Software Engineering*, **SE-13** (3), 1987, 311-323.

[5] T. J. Biggerstaff, B. G. Mitbander, D. E. Webster, Program understanding and the concept assignment problem, *Communications of the ACM*, **37** (5), 1994, 72-83.

[6] G. Canfora, A. Cimitile, M. Munro, A reverse engineering method for identifying reusable abstract data types, Proceedings of the Working Conference on Reverse Engineering, Baltimore, Maryland, May 1993, 73-82.

[7] G. Canfora, A. Cimitile, M. Munro, An improved algorithm for identifying objects in code, *Software - Practice and Experience*, **26** (1), 1996, 25-48.

[8] G. Canfora, A. Cimitile, M. Munro, C. J. Taylor, Extracting abstract data types from C programs: a case study, Proceedings of the Conference on Software Maintenance, Montreal, Canada, 1993, 200-209.

[9] G. Canfora, A. Cimitile, M. Munro, M. Tortorella, Experiments in identifying reusable abstract data types in program code, Proceedings of the 2nd Workshop on Program Comprehension (WPC'93), Capri, Italy, July 1993, 36-45.

[10] Y.-F. Chen, M. Y. Nishimoto, C. V. Ramamoorthy, The C information abstraction system, *IEEE Transactions on Software Engineering*, **16** (3), 1990, 325-334.

[11] E. J. Chikofsky, J. H. Cross II, Reverse engineering and design recovery: a taxonomy, *IEEE Software*, **7** (1), 1990, 13-17.

[12] A. Cimitile, U. De Carlini, Reverse engineering: algorithms for program graph production, *Software - Practice and Experience*, **21** (5), 1991, 519-537.

[13] L. Cleveland, A program understanding support environment, *IBM Systems Journal*, **28** (2), 1989, 324-344.

[14] P. Coad, Object-oriented patterns, *Communications of the ACM*, **35** (9), 1992, 152-159.

[15] P. Coad, E. Yourdon, *Object Oriented Analysis*, Yourdon Press Computing Series, Prentice Hall, Inc., Englewood Cliffs, 1991.

[16] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

[17] J. R. Cordy, N. L. Eliot, M. G. Robertson, TuringTool: a user interface to aid in the software maintenance task, *IEEE Transactions on Software Engineering*, **16** (3), 1990, 294-301.

[18] W. C. Dietrich, Jr., L. R. Nackman, F. Gracer, Saving a legacy with objects, OOPSLA'89 Conference Proceedings (Object-Oriented Programming: Systems, Languages and Applications), New Orleans, Louisiana, October 1989, appeared in: *Sigplan Notices*, **24** (10), 1989, 77-83.

[19] B. Freitag, A hypertext-based tool for large scale software reuse, in G. Wijers, S. Brinkkemper, T. Wasserman (eds.), *Advanced Information Systems Engineering*, Proceedings of the 6th International Conference, CAiSE'94, Utrecht, The Netherlands, June 1994, *Lecture Notes in Computer Science*, **811**, Springer, 1994, 283-296.

[20] H. Gall, R. Klösch, Capsule oriented reverse engineering for software reuse, in: I. Sommerville, M. Paul (eds.), Proceedings of the 4th European Software Engineering Conference (ESEC'93), Garmisch-Partenkirchen, Germany, September 1993, *Lecture Notes in Computer Science*, **717**, 418-433, Springer, 1993.

[21] H. Gall, R. Klösch, Program transformation to enhance the reuse potential of procedural software, ACM Symposium on Applied Computing (SAC'94), Phoenix, USA, March 1994. (Internet: `http://www.infosys.tuwien.ac.at/Staff/hg/sac94.ps`)

[22] H. Gall, R. Klösch, Managing uncertainty in an object recovery process, in: 5th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'94), July 1994, also in: B. Bouchon-Meunier, R. R. Yager, L. A. Zadeh (eds.), *Advances in Intelligent Computing*, Springer, 1995. (Internet: `http://www.infosys.tuwien.ac.at/Staff/hg/ipmu94.ps`)

[23] H. Gall, R. Klösch, E. Kofler, L. Würfl, Balancing in reverse engineering and in object-oriented systems engineering to improve reusability and maintainability, in: 18th IEEE Computer Software and Application Conference, COMPSAC'94, 1994. (Internet: `http://www.infosys.tuwien.ac.at/Staff/hg/compsac-cr.ps`)

[24] H. C. Gall, R. R. Klösch, R. T. Mittermeir, Architectural transformation of legacy systems, in: W. Griswold (ed.), 17th International Conference on Software Engineering (ICSE-17), Workshop on Program Transformation for Software Evolution, Technical Report Number CS95-418, Seattle, April 1995. (Internet: `http://www.infosys.tuwien.ac.at/Staff/hg/icse-ws.ps`)

[25] H. Gall, R. Klösch, R. Mittermeir, Object-oriented re-architecturing, in: W. Schäfer, P. Botella (eds.), Proceedings of the 5th European Software

Engineering Conference (ESEC'95), Sitges, Spain, September 1995, *Lecture Notes in Computer Science*, **989**, 499-519, Springer, 1995. (Internet: `http://www.infosys.tuwien.ac.at/Staff/hg/esec-cr.ps`)

[26] H. C. Gall, R. R. Klösch, R. T. Mittermeir, Application patterns in re-engineering: identifying and using reusable concepts, in: 6th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'96), vol. III, 1099-1106, July 1996. (Internet: `http://www.infosys.tuwien.ac.at/Staff/hg/ipmu-cr-web.ps`)

[27] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[28] D. Jackson, E. J. Rollins, A new model of program dependencies for reverse engineering, Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA, December 1994, in: *Software Engineering Notes*, **19** (5), 1994, 2-10.

[29] I. Jacobson, F. Lindström, Re-engineering of old system to an object-oriented architecture, OOPSLA'91 Conference Proceedings (Object-Oriented Programming Systems, Languages, and Applications), 1991, 340-350.

[30] S. Jarzabek, C. L. Tan, Modeling multiple views of common features in software reengineering for reuse, in: G. Wijers, S. Brinkkemper, T. Wasserman (eds.), *Advanced Information Systems Engineering*, Proceedings of the 6th International Conference, CAiSE'94, Utrecht, The Netherlands, June 1994, *Lecture Notes in Computer Science*, **811**, Springer, 1994, 269-282.

[31] R. R. Klösch, Reverse engineering: why and how to reverse engineer software, in: Proceedings of the California Software Symposium (CSS'96), University of Southern California, 92-99, April 1996. (Internet: `http://www.infosys.tuwien.ac.at/Staff/hg/css-cr-web.ps`)

[32] P. K. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, P. Tulula, Visualizing program dependences: an experimental study, *Software - Practice and Experience*, **24** (4), 1994, 387-403.

[33] S. S. Liu, N. Wilde, Identifying objects in a conventional procedural language: an example of data design recovery, Proceedings of the Conference on Software Maintenance, San Diego, California, November 1990, 266-271.

[34] P. E. Livadas, T. Johnson, A new approach to finding objects in programs, *Journal of Software Maintenance: Research and Practice*, **6**, 1994, 249-260.

[35] G. C. Murphy, D. Notkin, Lightweight lexical source model extraction, *ACM Transactions on Software Engineering and Methodology*, **5** (3), 1996, 262-292.

[36] H. Mössenböck, K. Koskimies, Active text for structuring and understanding source code, *Software - Practice and Experience*, **26** (7), 1996, 833-850.

[37] P. Newcomb, Reengineering procedural into object-oriented systems, Proceedings of the 2nd Working Conference on Reverse Engineering, Toronto, Canada, July 1995, 237-249. (a more abstract version of the paper can be found in Internet: `http://www.softwarerevolution.com/tsri/htrpioo.html`)

[38] J. Q. Ning, A. Engberts, W. Kozaczynski, Recovering reusable components from legacy systems by program segmentation, Proceedings of the Working Conference on Reverse Engineering, Baltimore, Maryland, May 1993, 64-72.

[39] J. Q. Ning, A. Engberts, W. Kozaczynski, Legacy code understanding, *Communications of the ACM*, **37** (5), 1994, 50-57.

[40] R. M. Ogando, S. S. Yau, S. S. Liu, N. Wilde, An object finder for program structure understanding in software maintenance, *Journal of Software Maintenance: Research and Practice*, **6**, 1994, 261-283.

[41] S. Paul, A. Prakash, A framework for source code search using program patterns, *IEEE Transactions on Software Engineering*, **20** (6), 1994, 463-475.

[42] T. Philip, R. Ramsundar, A reengineering framework for small scale software, *Software Engineering Notes*, **20** (5), 1995, 51-55.

[43] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, (Third Edition, European Adaptation), 1994.

[44] A. Quilici, A memory-based approach to recognizing programming plans, *Communications of the ACM*, **37** (5), 1994, 84-93.

[45] V. Rajlich, N. Damaskinos, P. Linos, W. Khorshid, Vifor: a tool for software maintenance, *Software - Practice and Experience*, **20** (1), 1990, 67-77.

[46] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Inc., Englewood Cliffs, 1991.

[47] H. M. Sneed, E. Nyáry, Extracting object-oriented specification from procedurally oriented programs, Proceedings of the 2nd Working Conference on Reverse Engineering, Toronto, Canada, July 1995, 217-226.

[48] E. B. Swanson, The dimensions of maintenance, Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, October 1976, 492-497.

[49] M. Weiser, Program slicing, *IEEE Transactions on Software Engineering*, **SE-10** (4), 1984, 352-357.

[50] N. Wilde, S. M. Thebaut, The maintenance assistant: work in progress, *The Journal of Systems and Software*, **9** (1), 1989, 3-17.