Recovering Class Diagrams from Data-Intensive Legacy Systems

Giuseppe Antonio Di Lucca, Anna Rita Fasolino, Ugo De Carlini
dilucca/ fasolino/ decarl @unina.it

Dipartimento di Informatica e Sistemistica - Università di Napoli Federico II
Via Claudio 21, 80125 Naples, Italy

Abstract

Several reverse engineering methods for recovering objects from legacy systems have been proposed in the literature, but most of them neglect to identify the relationships among the objects, or recover only a part of them. This paper describes a method for recovering an O-O model together with the objects and relationships among them. The proposed approach integrates the results of reverse engineering of both the procedural code and the persistent data stores of the system, and exploits a number of heuristic criteria to obtain a class diagram. A preliminary experiment carried out to validate the method on a COBOL medium-sized system yielded encouraging results.

1. Introduction

A large number of IT organisations have undertaken to migrate their legacy systems [1] towards open systems based on object-oriented technology and distributed client-server platforms. Migrating to Object-Oriented (O-O) technologies entails the definition of an O-O model of the application domain. One of the most immediate ways to obtain such a model may be based on an *ex-novo* object-oriented analysis of the domain. This model could describe all the relevant objects from the application domain, and their relationships.

Although feasible, such a solution may not be effective, as it does not take into account the domain knowledge and the expertise about the business rules, that are usually recorded only in the legacy code. Therefore, during the development of a replacement system, the knowledge encapsulated in the previous system should also be exploited.

An alternative solution to the problem of achieving an object-oriented model of the application domain is by reverse engineering the existing system. Several reverse engineering methods have been proposed for recovering objects from the code [5, 6, 7, 8, 9, 11, 12, 14, 15, 16, 17]. With these methods, objects can be searched for at different degrees of granularity: for instance, recovering coarse-grained objects can be centred around persistent data, whereas recovering fine-grained objects entails the selection of volatile data. The same methods also define approaches for recovering object operations, associating them with chunks of code at different granularity levels, such as programs, sub-routines, clusters of sub-routines, or program slices. For each of these approaches, user interaction is required to refine the initial results and assign a meaning to the objects identified.

The methods described in the literature mostly aim to recover objects from the code, but they do not usually address the problem of searching for the relationships among them and thus result in incomplete O-O models. Consequently, these methods are not effective for reverse engineering *data-intensive* systems, *i.e.* systems where much of the complexity lies in the data structures and in the relationships among them. Moreover, these approaches may not be practical when a knowledge of the relationships among objects is essential as a basis for designing the deployment of the objects on separate sites, as is usually required in a migration process towards distributed systems.

Other methods address the problem of recovering an O-O model together with the main relationships between objects. Some of them recover objects from files or database schemas, and the relationships between them are identified by analysing only the links established by the usage of key attributes and foreign keys defined in the file/database structures [3, 10]. However, such methods do not take into account data dependencies among data structures, that would have to be extracted by analysing the data-flow in the code, and so may result in incomplete O-O models.

A reverse engineering approach for recovering an object-oriented model from a non O-O legacy system is proposed in this paper. Since the approach produces the system class diagram, including both classes and their relationships, it can be successfully employed in

1063-6773/00 $10.00 © 2000 IEEE

migration processes, to obtain the initial target O-O model to be migrated towards new platforms. Analogously, the approach can be used in a replacement project to develop the preliminary requirements for a replacement system, ensuring that the features of the previous system are not overlooked or forgotten in the replacement one [2].

Recovering an O-O model from legacy systems is always a human-intensive process, where reverse engineers and their knowledge about the system and the application domain play the predominant role. The method proposed in this paper supports the work of reverse engineers by providing them with a set of *heuristic criteria.* The criteria are designed to identify candidate objects, their attributes, and relationships from *data-intensive systems*, that are strongly based on persistent data stores. The source code, the structure of the data stores, and the data flow are all analysed and taken into account by the heuristic criteria. More precisely, the criteria allow coarse-grained objects to be identified from persistent data stores, and their relationships to be defined by analysing the actual implementation strategies most commonly encountered in traditional systems for implementing relationships between related data structures. The criteria do not address the recovery of object methods, since other reverse engineering approaches described in the literature can be used for this purpose.
The method has been validated in a preliminary experiment carried out on a medium-sized COBOL system. The results of the experiment are presented in the paper.

The paper is structured as follows: section 2 illustrates the target O-O model and the data-intensive systems addressed in the paper. The proposed method is described in section 3, while section 4 illustrates the application of the method to a case study. Some conclusive remarks are made in section 5.

2. The Context

2.1. The target Object-Oriented Model

An Object-Oriented model of a given application domain includes a set of *classes*, representing the data abstractions from the application domain. Each class consists of a template to be used for instantiating *objects*, and encapsulates a set of attributes and a set of operations: the object's encapsulated attributes define the *state* of the object, while the state and the operations together define the object's *behaviour.*

An O-O model includes *associations* among classes, which describe both the structural relationships among the instances of the classes, and the usage relationships among them (client/server relationship). A peculiar characteristic of an O-O model is the *generalisation-specialisation* relationship, which is used by a class to inherit all or some of the attributes and services of a super-class, and to add more attributes and services.

Another kind of relationship among classes is the *whole-part* one, representing the structural relationship between a whole and its parts.

The O-O model to be recovered from data-intensive systems can be described by a class diagram including *classes*, *associations*, *generalisation-specialisation,* and *whole-part* relationships, according to the UML [4].

2.2. Data-intensive systems

A data-intensive system usually processes a large volume of persistent data, which typically correspond to the application domain elements whose values are relevant for the organisation's business goals. Such elements are usually stored in *files* in the file-system, or database *tables* from which they can be retrieved for processing.

The persistent data can be organised in data structures, such as the records of files, or the schemas of database tables. These data structures represent every single relevant element from the application domain. In the following, the term *record* will be used to refer to such data structures.

The ways the elements from the application domain interact, or are logically related, can be expressed in the code in terms of peculiar relationships among the corresponding records. Different relationships can be implemented in different ways, depending on the implementation mechanisms available, and on the specific interaction to be realised. The approach presented in this paper is based on the recovery of such interaction mechanisms.

3. Recovering the O-O model

The approach proposed in this paper for recovering an O-O model from a data intensive system preliminarily assumes each record description to be a candidate class, and record instances to be the objects of that class. Record fields constitute the class attributes, while programs provide the class operations, according to the method described in [7]. The relationships among candidate classes will be deduced by analysing both the record fields and their data-flow in the code. The proposed method provides a number of suitable heuristics for accomplishing this task.

Recovery of the O-O model is thus achieved by means of the following steps:

1 Identification of candidate classes;
2 Identification of associations among candidate classes;
3 Identification of generalisation-specialisation relationships;
4 Identification of whole-part relationships;
5 Validation of the model.

These steps are illustrated in the sub-sections below.

3.1. Identification of candidate classes

The first step in the method consists of the identification of the persistent data-stores managed by the system: each record, which is associated with a system data-store, is assumed to be a candidate class. Each class will be given the name of the associated record.

The existence of synonyms and homonyms between records compounds the difficulty of identifying candidate classes and their attributes. Two records are *synonymous* if they have different names but the same record structure, and thus correspond to the same element in the application domain. The descriptions of synonymous record structures may even be different, although such differences just consist of different levels of decomposition of the record fields. Two records are *homonymous* if they have the same name but a different record structure, and thus correspond to different elements in the application domain. In practice, homonymy is usually due to the usage of the same data store name in different programs, each of which defines a different record structure for a different data store.

During the first step, synonymous records should be associated with one and the same candidate class, as they may correspond to the same concept of the application domain. On the contrary, homonymous records should be associated with different candidate classes, as they may represent different concepts in the application domain. In both cases, a meaningful name has to be assigned to each candidate class.

The *attributes* of each candidate class can be defined as the fields of the record associated with the class. If synonymous records have been identified, and all these records have the same fields (*i.e.* the same names and structures), these fields will be assumed to be the class attributes. If, on the other hand, the number, name, and/or structure of the record fields are different, the class attributes will be defined as the record fields with the most detailed degree of granularity. In both cases, class attributes have to be given meaningful names.

A concept from the application domain should be assigned to each candidate class: this task requires the analysis of the available software documentation. Candidate classes that do not represent meaningful abstractions from the application domain will be discarded.

3.2 Identification of *associations* among classes

The identification of association relationships among classes requires the analysis of both the structure of the records, and the data-flow involving their fields in the code. The aim of this analysis is to uncover those mechanisms usually employed in non-O-O programming languages to implement relationships among different records. For instance, relationships among two or more records typically take place through:

- *common fields* (particularly fields composing the record identifiers), *i.e.* the same attributes replicated in different records;
- *correspondence tables*, *i.e.* tables including the identifiers of different records;
- *data dependencies* among attributes belonging to different records.

Candidate association relationships between classes can be defined by applying the following heuristic criteria. Human intervention will then be required to validate the candidate associations, to assign them a concept, and to identify their direction and multiplicity. The knowledge about the application domain and the software engineer's expertise will be necessary for accomplishing this task.

Prior to applying these criteria, the record identifiers must be preliminarily recovered.

Criterion A1

Identify couples of records having common fields, and define a candidate association between the classes corresponding to the records involved. Common fields that constitute record identifiers should be searched for particularly, and used to define an association. Common fields are not required to have the same name, but it is essential that they have the same meaning.

Criterion A2

Identify each record whose fields can be partitioned into subsets, each of which corresponds to the identifiers of other records that do not have any attributes in common. A candidate association between the classes associated with the latter records can be defined.

This criterion exploits the correspondence tables mechanism used for implementing a relationship between records.

If *n* different classes are involved in a relationship through a correspondence table, this criterion will indicate an *n-ary* association among the *n* classes.

Of course, after applying this criterion, the class associated with the correspondence table will be discarded from the set of candidate classes.

Criterion A3

Identify different records that are data-dependent on some fields, and define a candidate association between the classes associated with these records.

This association is more likely if the fields involved in the data dependency include those that constitute the record identifiers.

Criterion A4

Identify different records whose fields are included in one and the same input/output form/report, and define a candidate association among the classes corresponding to these records.

The likelihood of this association depends strongly on the degree of correlation of the fields included in the form/report. In this case, human intervention is needful to decide whether the association actually exists.

Criterion A5

Identify different records associated with the same file (*i.e.* a file with multiple records), and define a candidate association between the classes corresponding to these records.

Given an association between two classes, the association itself may have properties that can be modelled by an association class [4]. The following criterion *AC1* suggests how association classes can be identified.

Criterion AC1

An association class between two associated classes is likely to exist if a third class, exclusively associated with the former ones, can be found. The likelihood of such an association class existing is greater if it includes attributes corresponding to the record identifiers of the other classes. The knowledge and expertise of the software engineer will be needed to decide whether the association class actually exists.

The candidate relationships obtained by applying the criteria listed above will have to be validated on the basis of the software engineer's knowledge and expertise in the application domain.

3.3 Identification of *generalisation-specialisation* relationships

The identification of generalisation-specialisation (Gen-Spec) relationships between a general class (the super-class) and a more specialised one (a sub-class) exploits the methods usually employed for implementing this kind of abstraction in procedural languages. For instance, these methods comprise:

- including both the super-class attributes and the sub-classes attributes in one and the same record, adding an attribute whose values discriminate among the sub-classes;
- employing as many records as there are sub-classes, and including the super-class attributes besides the sub-class attributes in each record;
- employing one record for the super-class and as many records as there are sub-classes. The record implementing the super-class will be related with each record implementing a sub-class either by common fields or by correspondence tables.

Gen-Spec relationships between classes can be defined by applying the heuristic criteria listed below. Some criteria apply to the association relationships recovered and aim to assess if they can be qualified as Gen-Spec relationships. In this case, the association must be discarded from the model and replaced with a suitable Gen-Spec relationship.

In the following criteria, the terms classes and records will be used as synonymous, as will attributes and fields.

Criterion Gen-Spec1

Identify a class including, besides other attributes, different attribute sub-sets whose instances cannot be defined simultaneously, but assume values depending on the value of a discriminating attribute (or group of attributes). This criterion suggests defining a hierarchy with a different sub-class including each different attribute sub-set, and a super-class including the remaining common attributes.

As an example, consider a record including the generic attributes describing a publication (*i.e.* a book or a journal), book specific attributes, and journal specific ones, besides an attribute, 'type-of-publication', assuming only two values. Depending on the value of this attribute, either the book or the journal attributes will be defined in each record instance. In such a case, the criterion suggests defining a super-class (*i.e.* the publication class) including the set of generic attributes (*i.e.* the publication attributes) and a different sub-class containing the attributes for each specific type of publication (*i.e.* the book and journal classes).

Criterion Gen-Spec2

Identify classes that, having sets of common attributes besides disjointed ones, are linked by an association relationship (see criterion A1). This criterion suggests defining a candidate hierarchy with a super-class including the common attributes, and a different sub-class for each class that includes only the remaining attributes in each. In this case, the engineer's expertise and knowledge of the application domain will be used to establish the most adequate relationship between the Gen-Spec and the association. The set of common attributes should not only include attributes corresponding to the record identifiers (or part of them) because in such a case, an association rather than a generalisation-specialisation relationship might exist.

Criterion Gen-Spec3

Identify classes having all their attributes in common (*i.e.* all identical attributes): if they were not clustered previously because they were not synonymous, it will be necessary to check whether they are actually separate classes, or belong to a generalisation-specialisation hierarchy where the sub-classes have no more attributes other than those of the super-class.

In such a case, the criterion suggests defining a hierarchy, with a super-class including the set of common attributes, and a sub-class for each class including no other attributes than the super-class.

Criterion Gen-Spec4

Identify classes including only attributes constituting the identifiers of other classes (*i.e.* the identifiers of the records associated with these classes). This kind of class is used to implement a correspondence between other classes. In such a case, it will be necessary to check and decide, on the basis of the software engineer's expertise and knowledge of the application domain, whether the correspondence indicates only an association (see criterion A2), or a Gen-Spec relationship between classes. In the latter case, a suitable hierarchy will have to be defined.

After applying these criteria, the associations between sub-classes involved in a hierarchy should be assessed to decide whether they should be discarded or not. As a general rule, an association due to common attributes allocated in the super-class should be removed, while an association due to the attributes belonging to the sub-classes should be preserved.

3.4 Identification of *whole-part* relationships

Abstracting *whole-part* relationships from the code may be a difficult task, since both the association and the whole-part relationship may be implemented by the same mechanisms in the code. In this case, the software engineer's knowledge and expertise in the application domain is essential to distinguish the whole-part relationships.

The semantics of a whole-part relationship can be characterised with reference to three main features: the *lifetime dependence*, the *degree of dependence* and the *degree of sharing* among the 'whole' object and its 'part' objects [13]. The lifetime dependence indicates that the part objects are born and/or die together with the whole object; the dependence degree specifies whether the part objects may exist independently of the whole object or not; finally, the degree of sharing specifies whether the part objects are entirely owned by the whole objects, or else may be shared among other objects. The following criteria exploit this characterisation to identify whole-part relationships.

Criterion Wp1

Given a set of classes linked by an association relationship and the associated records, if analysis of the procedural code shows that each time a new instance of a record is created (*i.e.* written), new instances of the other records are created too, then a candidate whole-part relationship can be defined between these classes.

Criterion Wp2

Given a set of classes linked by an association relationship and the associated records, if analysis of the procedural code shows that each time an instance of a record is deleted, instances of the other records are deleted too, then a candidate whole-part relationship can be defined between these classes.

Criterion Wp3

Given a set of classes linked by an association relationship and the associated records, if analysis of the procedural code shows that each time a reference to a record instance is made, a reference to other record instances (one or more) is also executed, then a candidate whole-part relationship can be defined between these classes.

The UML subdivides the whole-part relationship into the categories of *aggregation* and *composition* relationships. The following general criteria provide an approach for recovering these.

Criterion C1

If two or more classes satisfy the Wp1 and Wp2 criteria simultaneously, a lifetime dependence between

these classes can be deduced, and a candidate *composition* relationship between them can be defined.

Criterion C2

If two or more classes satisfy the Wp1 and Wp3, or the Wp2 and Wp3 criteria simultaneously, a lifetime dependence between these classes can be deduced, and a candidate *composition* relationship between them can be defined.

Criterion C3

If two or more classes satisfy at least one of the Wp1, Wp2, Wp3 criteria, so that a whole-part relationship between them has been deduced, and the 'whole' class objects are the exclusive owners of the 'part' objects (*i.e.* the part objects are not shared with other objects), a candidate *composition* relationship between the classes can be defined.

Criterion C4

If two or more classes do not satisfy any of the C1, C2, C3 criteria, but they satisfy at least one of the Wp1, Wp2, Wp3 criteria, and the 'whole' class objects are not the exclusive owners of the 'part' objects (*i.e.* the part objects are shared with other objects), a candidate *aggregation* relationship between the classes can be defined.

3.5 Validation of the O-O model

The seventeen criteria illustrated above enable definition of a preliminary class diagram including candidate classes and relationships. The initial diagram should be submitted to a validation process, to assess the adequacy of each class and relationship it includes. The validation process has to verify that each recovered class, association, Gen-Spec, and whole-part relationship represents a meaningful abstraction from the application domain; moreover, association and Gen-Spec relationships must not overlap.

Both the software engineer's experience and the knowledge of the application domain will be required to validate the model.

4. A Case-Study

A preliminary experiment was carried out to validate the method proposed in the paper. An information system written in COBOL was used as a case study, and its class diagram was recovered using the method. The recovered diagram was validated, and the validation results were used to obtain preliminary indications about the validity of the method. In practice, the effectiveness of the method was assessed on the basis of the number of candidate classes and relationships discarded from the initial model.

The main characteristics of the system, the tools employed during the experiment, the experimental procedure adopted, and the results obtained are described below. To conclude, the validity of the method is discussed with reference to the validation results.

4.1 Experimental materials and tools

An information system written in COBOL, comprising 103 programs and 90 copybooks, of an overall size of about 200 KLOC, was chosen for the experiment. The software system managed a University hall and residence and was designed by adopting a functional decomposition approach. Different functional requirements were implemented by different subsystems.

A commercial tool was used for the static analysis of the source code to extract the information required by the method. The static analyser stored the analysis results in a proprietary repository, that allowed only predefined queries and reports to be produced. To overcome this limit, the information about the data stores and the record structures provided by the tool was exported to a relational database, from which the further information required could be obtained.

4.2 Experimental procedure and results

The procedure for recovering the O-O model required four initial steps, during which the heuristic criteria were applied and their results were preliminarily validated. In a fifth step, the recovered class diagram was submitted to a validation process.

The description of the above steps and the results obtained are reported below.

1. Recovering candidate classes

In the first step, the source code was statically analysed to identify its *persistent* data stores and the associated records. The subject system employed only files as persistent data stores, which were identified by analysing both the *input-output section* of the *environment divisions,* and the *file section* of the *data divisions*. During the analysis, files assigned with mass-storage devices, such as disks, were searched for, while files assigned with other I/O devices, such as printers, were ignored. This analysis yielded 53 disk files, whose names are listed in Table 1.

Files that did not represent meaningful abstractions from the application domain were searched for in the list. Temporary files, files used only in sort/merge operations,

Table 1: The system data stores

ACCOUNTS.LIS	OUTSCR.TMP	_PROCESS.DAT
AHISTORY.AUD	PAYMENT.AUD	_REQUEST.DAT
ARCHIVE.ARC	REPAY.AUD	_RECEIPT.TMP
ARCHIVE.TMP	SELECT.TMP	_ROOM.DAT
CASH.DAT	SORT.TMP	_ROOMINV.DAT
CHARGES.AUD	UNSORT.TMP	_SORT.TMP
CHARGES.DAT	_BATCH.DAT	_USAGE.DAT
COLLEGE.GNT	_CITIZEN.DAT	_USER.DAT
CONFIG.DAT	_DAMAGE.DAT	BACKUP-FILE
CONIN.AUD	_DUTY.DAT	CONFIG-FILE
CONOUT.AUD	_DEPART.DAT	DIARY-FILE
CREDIT.AUD	_FACULTY.DAT	DIARY2-FILE
DEPOSIT.AUD	_HISTORY.DAT	ERROR-FILE
HARRIS.LBT	_IN.TMP	INSTALL-FILE
INSCR.TMP	_ITEMINV.DAT	JOURNAL-FILE
MASTER.DAT	_LAST.DAT	TEXT-FILE
NAMES.TMP	_OUT.TMP	ZLIST-FILE
NONCASH.DAT	_PREFER.DAT	

Table 2: The system data stores after synonymous/homonymous analysis

ACCOUNTS.LIS	NONCASH.DAT	_PROCESS.DAT
AHISTORY.AUD	PAYMENT.AUD	_REQUEST.DAT
ARCHIVE.ARC	REPAY.AUD	_RECEIPT.TMP
CASH.DAT	_CITIZEN.DAT	_ROOM.DAT
CHARGE.AUD	_DAMAGE.DAT	_ROOMINV.DAT
CHARGES.DAT	_DUTY.DAT	_USER.DAT
CONIN.AUD	_DEPART.DAT	DIARY-FILE
CONOUT.AUD	_FACULTY.DAT	DIARY2-FILE
CREDIT.AUD	_HISTORY.DAT	JOURNAL-FILE
DEPOSIT.AUD	_ITEMINV.DAT	
MASTER.DAT	_PREFER.DAT	

Table 3: The classes associated with the data store records

_CITIZEN-RECORD	CREDIT-RECORD	PAYMENT-RECORD
IN-CONTROL-RECORD	DAMAGE-RECORD	PREFERENCES-RECORD
OUT-CONTROL-RECORD	DEPOSIT-RECORD	PROCESSED-ROOM
_DEPART-RECORD	DIARY-RECORD	RECEIPT-RECORD
ACCOUNTS-HEADER	DUTY-RECORD	REPAY-RECORD
ACCOUNTS-RECORD	FACULTY-RECORD	REQUEST-RECORD
ARCHIVE-RECORD	HISTORY-RECORD	ROOM-RECORD
AUDIT-HISTORY-RECORD	ITEMINV-RECORD	ROOMINV-RECORD
CASH-RECORD	JOURNAL-RECORD	USER-RECORD
CHARGE-RECORD	MASTER-RECORD	
CHARGES-RECORD	NONCASH-RECORD	

files just storing information about the system installation/configuration, and files storing the error messages were found and discarded. After this refinement step, the 31 files shown in Table 2 were obtained. Among them, the temporary file _RECEIPT.TMP was not discarded, due to its relevance in the application domain: it represented the payment receipt given to a college resident. This file was a temporary one, because it was removed from the file system once the receipts it stored had been printed.

The file list was further assessed to detect synonymous and homonymous files. Just two synonymous files, namely DIARY-FILE and DIARY2-FILE, were found (and their records were associated with a same class). Finally, a file with multiple records was identified, the file ACCOUNTS.LIS, that included the records ACCOUNTS-HEADER and ACCOUNTS-RECORD, which were associated with two distinct classes.

After these refinement steps, records associated with the resulting files were assumed to be candidate classes. The 31 identified classes are reported in Table 3 with the same names as the associated record.

At this point, record identifiers were searched for. With respect to the *indexed* and *relative* file records (13 *indexed* and 1 *relative* file were managed by the system), the fields declared as *record key*, *alternate record key*

Table 4: The associations identified between classes

CLASS	CLASS	Criterion
ACCOUNTS-RECORD	ACCOUNTS-HEADER	A5
ACCOUNTS-RECORD	CHARGE-RECORD	A3
ACCOUNTS-RECORD	CHARGES-RECORD	A3
ACCOUNTS-RECORD	CREDIT-RECORD	A3
ACCOUNTS-RECORD	DAMAGE-RECORD	A3
ACCOUNTS-RECORD	MASTER-RECORD	A1
ACCOUNTS-RECORD	PAYMENT-RECORD	A3
ACCOUNTS-RECORD	REPAY-RECORD	A3
AUDIT-HISTORY-RECORD	USER-RECORD	A1
CASH-RECORD	CHARGE-RECORD	A1
CASH-RECORD	CREDIT-RECORD	A1
CASH-RECORD	MASTER-RECORD	A1
CASH-RECORD	REPAY-RECORD	A1
CASH-RECORD	ROOM-RECORD	A1
CHARGE-RECORD	CREDIT-RECORD	A1
CHARGE-RECORD	JOURNAL-RECORD	A3
CHARGE-RECORD	MASTER-RECORD	A1
CHARGE-RECORD	REPAY-RECORD	A1
CHARGE-RECORD	ROOM-RECORD	A1
CHARGES-RECORD	MASTER-RECORD	A1
CREDIT-RECORD	JOURNAL-RECORD	A3
CREDIT-RECORD	MASTER-RECORD	A1
CREDIT-RECORD	ROOM-RECORD	A1
DEPOSIT-RECORD	CASH-RECORD	A3
DEPOSIT-RECORD	CHARGE-RECORD	A1
DEPOSIT-RECORD	CREDIT-RECORD	A1
DEPOSIT-RECORD	JOURNAL-RECORD	A3
DEPOSIT-RECORD	MASTER-RECORD	A1
DEPOSIT-RECORD	NONCASH-RECORD	A3
DEPOSIT-RECORD	REPAY-RECORD	A1
DEPOSIT-RECORD	ROOM-RECORD	A1
DIARY-RECORD	DUTY-RECORD	A3
HISTORY-RECORD	ROOM-RECORD	A3
IN-CONTROL-RECORD	ROOM-RECORD	A1
ITEMINV-RECORD	ROOMINV-RECORD	A1
ITEMINV-RECORD	ROOM-RECORD	A1
JOURNAL-RECORD	ACCOUNTS-RECORD	A3
JOURNAL-RECORD	CHARGES-RECORD	A3
JOURNAL-RECORD	DAMAGE-RECORD	A3
JOURNAL-RECORD	MASTER-RECORD	A3
JOURNAL-RECORD	PAYMENT-RECORD	A3
MASTER-RECORD	_CITIZEN-RECORD	A1
MASTER-RECORD	_DEPART-RECORD	A1
MASTER-RECORD	ARCHIVE-RECORD	A1
MASTER-RECORD	FACULTY-RECORD	A1
MASTER-RECORD	IN-CONTROL-RECORD	A1
MASTER-RECORD	PREFERENCES-RECORD	A1
MASTER-RECORD	REQUEST-RECORD	A1
MASTER-RECORD	ROOM-RECORD	A1
NONCASH-RECORD	CHARGE-RECORD	A1
NONCASH-RECORD	CREDIT-RECORD	A1
NONCASH-RECORD	MASTER-RECORD	A1
NONCASH-RECORD	REPAY-RECORD	A1
NONCASH-RECORD	ROOM-RECORD	A1
OUT-CONTROL-RECORD	ACCOUNTS-RECORD	A3
OUT-CONTROL-RECORD	MASTER-RECORD	A1
OUT-CONTROL-RECORD	ROOM-RECORD	A1
PAYMENT-RECORD	CASH-RECORD	A3
PAYMENT-RECORD	CHARGE-RECORD	A1
PAYMENT-RECORD	CREDIT-RECORD	A1
PAYMENT-RECORD	DEPOSIT-RECORD	A1
PAYMENT-RECORD	MASTER-RECORD	A1
PAYMENT-RECORD	NONCASH-RECORD	A3
PAYMENT-RECORD	REPAY-RECORD	A1
PAYMENT-RECORD	ROOM-RECORD	A1
PREFERENCES-RECORD	PROCESSED-ROOM	A1
PREFERENCES-RECORD	REQUEST-RECORD	A1
PREFERENCES-RECORD	ROOM-RECORD	A1
PROCESSED-ROOM	ROOM-RECORD	A1
RECEIPT-RECORD	ACCOUNTS-RECORD	A3
RECEIPT-RECORD	CASH-RECORD	A3
RECEIPT-RECORD	DEPOSIT-RECORD	A3
RECEIPT-RECORD	JOURNAL-RECORD	A3
RECEIPT-RECORD	MASTER-RECORD	A1
RECEIPT-RECORD	NONCASH-RECORD	A3
RECEIPT-RECORD	PAYMENT-RECORD	A3
RECEIPT-RECORD	ROOM-RECORD	A1
REPAY-RECORD	CREDIT-RECORD	A1
REPAY-RECORD	JOURNAL-RECORD	A3
REPAY-RECORD	MASTER-RECORD	A1
REPAY-RECORD	ROOM-RECORD	A1
REQUEST-RECORD	ROOM-RECORD	A1
ROOMINV-RECORD	ROOM-RECORD	A1
ROOM-RECORD	DAMAGE-RECORD	A1
USER-RECORD	DIARY-RECORD	A1
USER-RECORD	DUTY-RECORD	A1
USER-RECORD	MASTER-RECORD	A3
USER-RECORD	RECEIPT-RECORD	A1
USER-RECORD	ROOM-RECORD	A3

and *relative key* in the *select* statements from the *input-output sections* were used as record identifiers.

Finally, the methods associated with these classes were recovered using the approach described in [7].

2. Recovering association relationships

The five criteria illustrated in section 3.2 allowed candidate association relationships to be defined among the classes. As far as *criterion A1* was concerned, it required querying the database to search for couples of records with common fields (and in particular common fields corresponding to record identifiers). A relationship was defined between the classes involved in each couple. This query resulted in 58 associations. No relationship was found when applying *criterion A2*, since no record implemented correspondence tables. Thanks to the facilities offered by the static analyser, data dependencies among the attributes of different classes were searched for (*criterion A3*). In this case, an association was defined between couples of classes whose attributes were linked by a data dependency. This criterion produced 30 additional associations between classes. Finally, *criterion A4* did not produce any new relationship, while another relationship was found thanks to *criterion A5*.

The final results are illustrated in Table 4, that shows 89 couples of classes linked by an association, and, for each

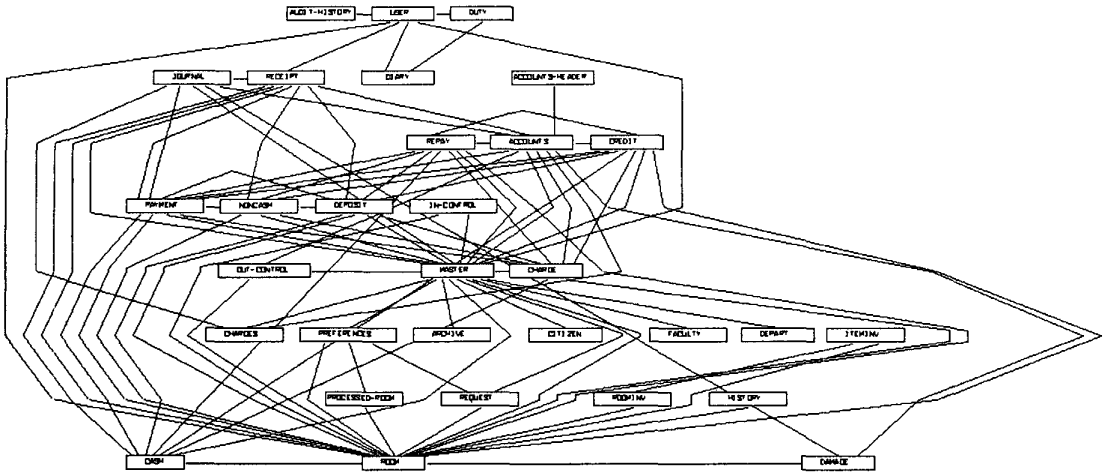


Figure 1: The initial Class Diagram

association, the criterion that produced it.

Figure 1 shows the initial class diagram representing all the classes and the associations found (in the diagram the suffix *–record* in each class name is not shown). The diagram shows how the system is centered on the `Master` class, which is responsible for college residents information and operations, and the `Room` class, responsible for college rooms information and operations. These classes are involved in most relationships (the `Master` class is involved in 22 relationships and the `Room` class in 19). The remaining classes are responsible for information and operations concerning:

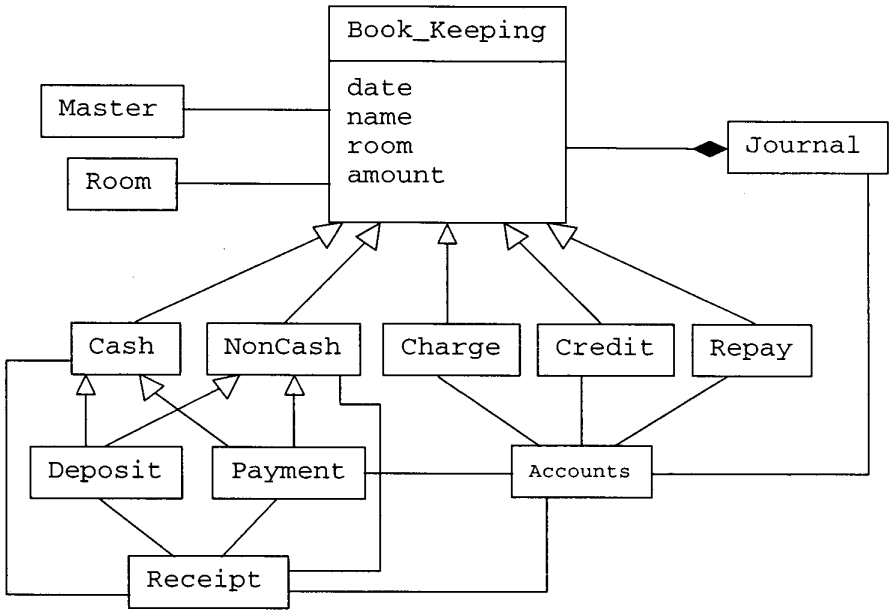
- Room booking, confirmation, check-in and check-out (classes `REQUEST`, `PREFERENCES`, `PROCESSED-ROOM`, `IN-CONTROL`, `OUT-CONTROL`)
- Room decoration, inventory, charges for damage and resident housekeeping duty (classes `ITEMINV`, `ROOMINV`, `DAMAGE`, `USER`, `DUTY`, `DIARY`)
- Financial administration (classes `DEPOSIT`, `PAYMENT`, `RECEIPT`, `CHARGES`, `ACCOUNTS`, `ACCOUNTS-HEADER`, `CREDIT`, `REPAY`, `JOURNAL`, `CHARGE`, `CASH`, `NONCASH`)
- Auditing and statistics (classes `AUDIT-HISTORY`, `HISTORY`, `ARCHIVE`)

Other classes are responsible for general information about residents' citizenship, Faculties and Departments (classes `CITIZEN`, `FACULTY`, `DEPART`), while the `ACCOUNTS-HEADER` class is a *singleton* class and stores pointers to file records of residents' accounts.

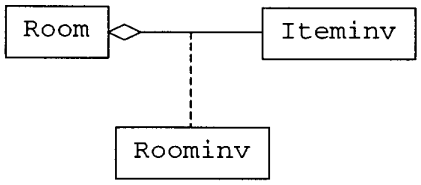
3. Identifying generalisation/ specialisation relationships

Gen-Spec2 criterion allowed a set of common attributes among the classes `Payment`, `Charge`, `Credit`, `Repay`, `Deposit`, `Cash` and `NonCash` to be identified; these classes represented financial matters, such as rent charging or payment made. A Gen-spec hierarchy among these classes was introduced, by grouping together the common attributes (`date`, `name`, `room`, `amount`) in a new class named `Book_Keeping` that became the super-class of the hierarchy. The *Gen-Spec2 criterion* also allowed two inner Gen-Spec structures to be recognised: a first structure involved the `Cash`, `NonCash` and `Payment` classes, and a second one the `Cash`, `NonCash` and `Deposit` classes, that recorded how a payment or a deposit were made (by cash or not): in the resulting hierarchies, the class `Payment`, as well as the class `Deposit`, inherit from the classes `Cash` and `NonCash` (see Figure 2-(a)).

All the associations among the `Payment`, `Charge`, `Credit`, `Repay`, `Deposit`, `Cash` and `NonCash` classes, previously defined, were removed and replaced with the new Gen-Spec relationships, while all the other associations these classes had with other classes were reorganised: in particular the relationships all the



(a)



(b)

Figure 2: The class diagram portions representing the Gen-Spec and Whole-part relationships

sub-classes had with the Master and Room classes were referred to the Book_Keeping super-class, while the ones which were peculiar to each sub-class were left unchanged. No further Gen-Spec relationships were recognised when the remaining Gen-Spec criteria were applied.

4. Identifying whole-part relationships

The association relationships were then again analysed according to the criteria defined in section 3.3, , to identify potential whole-part relationships between classes.

Wp1 criterion allowed a composition relationship between Book_Keeping and Journal classes to be defined: the latter consisted of the financial matters represented by the Book_Keeping class.

C4 criterion allowed an aggregation relationship between the Room and the Iteminv classes to be recognised: the latter class represents the pieces of furniture belonging to the rooms, thus a room is considered to consist of the iteminv it contains.

Moreover, the class Roominv was recognised as an association class (*criterion A1*): it indicated the number of each iteminv contained in the rooms (see Figure 2-(b)).

The class diagram shown in Figure 1 was therefore modified, according to the additional relationships recovered.

5. Validating the class diagram

As a conclusive step of the experiment, a final concept assignment process was carried out to validate the complete class diagram. The software documentation available, including the system user guide, and the knowledge of a domain expert were used in this phase. The validation process produced the following results: four classes (AUDIT-HISTORY, HISTORY, ARCHIVE, PROCESSED-ROOM) were discarded from the model, because they did not represent meaningful abstractions from the domain. The former three classes were associated with records of files storing statistical data, while the latter was associated with a file including just summary information used for room processing. As a consequence, the relationships between these classes and the others in the diagram were also discarded.

Finally, all the remaining classes and relationships were assigned a concept belonging to the application domain.

4.3 Assessing the effectiveness of the method

The case study results were used for preliminary assessment of the effectiveness of the proposed method.

The effectiveness was expressed by the *adequacy* of the method, that is the method's property to recover abstractions (both classes and relationships) that can be assigned a meaningful concept from the application domain. The following simple measure of *adequacy* was used:

$$Adequacy = \frac{|N|}{|M|} \times 100\,\%$$

where M is the set of abstractions selected by the method, N is the subset of components in M that can be associated with a concept of the application domain, and $|M|$ and $|N|$ denote the number of components in the M and N sets.

With respect to the recovered classes, the adequacy yielded a value of 87%, since 31 candidate classes were recovered by the method, while 27 were validated. As to the associations, the adequacy yielded a value of 93% (*i.e.* 89 candidate associations and 83 validated ones), while with the Gen-spec and whole-part relationships, the adequacy was equal to 100% in both cases.

The high values for adequacy obtained showed that the method was very effective. Of course, given the exploratory nature of the case study, this result cannot be generalised. A wider experiment, involving more than one system, and different application domains, should be designed and carried out to extend the validity of this experiment.

5. Conclusions

Recovering a class diagram from a non O-O legacy system is a preliminary step in most migration processes towards O-O distributed technologies. The relationships among the objects making up the diagram play an important role when designing the deployment of various system parts (*i.e.* the objects) in a distributed architecture, as is usually required in a migration process towards distributed platforms.

A reverse engineering method for recovering an Object-Oriented model from a non O-O legacy system has been proposed in the paper. The method focuses on data-intensive systems strongly based on persistent data stores, and employs seventeen heuristic criteria as a means for recovering the classes and relationships composing the system class diagram.

A preliminary experiment has been carried out to validate the proposed approach. A COBOL system about 200 KLOC in size was used as the case study, and its class diagram was recovered using the method. Finally, the diagram was validated by a concept assignment process, and the validation results were used to obtain preliminary indications about the validity of the method.

The method's effectiveness was assessed on the basis of the adequacy of the abstractions recovered (both classes and relationships). The high values for adequacy obtained (about 90%) show how effective the method is. However, other aspects of effectiveness, such as the completeness of the model recovered and the precision of the abstractions recovered, should be taken into account in a wider validation experiment. These will be addressed in future work.

Another point requiring further investigation is application of the approach in a migration process. The migration strategy should also aim to allocate the objects on the basis of the coupling deriving from the various relationships existing among them. Hence, the coupling between the objects of the recovered model should be assessed and exploited in the migration project. The selection of suitable metrics for evaluating this coupling, and the definition and validation a migration strategy guided by coupling measurements will also be addressed in future work.

References

[1] K. Bennett, "Legacy Systems: Coping with Success", IEEE Software, Jan. 1995, pp. 19- 23

[2] Blaha, M. R., and Premerlani, W. J. (1995) 'Observed Idiosyncrasies of Relational Databases Design' Proc. of 2nd *Working Conference on Reverse Engineering*, Toronto, Canada, IEEE Computer Society Press, 116-125

[3] M. Blaha, "An Industrial Example of Database Reverse Engineering", *Proc. of 6th Working Conference on Reverse Engineering*, IEEE CS Press, Los Alamitos, CA, 1999, pp. 196- 203

[4] G. Booch, J. Rumbaugh, I. Jacobson, *'The Unified Modeling Language User Guide'*, Addison Wesley, 1999

[5] P.T. Breuer, H. Haughton, and K. Lano, "Reverse-engineering COBOL via formal methods", *J. of Software Maintenance: Research and Practice*, vol. 5, 1993, pp. 13-35

[6] G. Canfora, A. Cimitile, and M. Munro, "An improved algorithm for identifying reusable objects in code", *Software Practice and Experiences*, vol. 26, no. 1, 1996, pp. 24-48

[7] A. Cimitile, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino, "Identifying objects in legacy systems using design metrics", *The Journal of Systems and Software*, vol. 44, January 1999, pp. 199-211.

[8] H. Gall and R. Klösch, "Finding objects in procedural programs: an alternative approach", *Proc. of 2nd Working Conference on Reverse Engineering*, Toronto, Canada, 1995, IEEE CS Press, pp. 208-216

[9] J. George and B.D. Carter, "A strategy for mapping from function oriented software models to object oriented software models", *ACM Software Engineering Notes*, vol. 21, no. 2, March 1996, pp. 56-63

[10] U. Kölsch, "Object-Oriented Re-engineering of Information Systems in a Heterogeneous Distributed Environment', *Proc. of 5th Working Conference on Reverse Engineering*, IEEE CS Press, Los Alamitos, CA, 1998, pp. 104- 114

[11] S. Liu and N. Wilde, "Identifying objects in a conventional procedural language: an example of data design recovery", *Proc. of Conference on Software Maintenance*, San Diego, CA, 1990, IEEE CS Press, pp. 266-271.

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

[13] R. Motschnig-Pitrik, J. Kaasboll, 'Part-Whole Relationship Categories and Their Application in Object-Oriented Analysis', IEEE Transactions on Knowledge and Data Engineering, vo. 11, no. 5, Sept/Oct. 1999

[14] P. Newcomb and G. Kotik, "Reengineering procedural into object-oriented systems", *Proc. of 2nd Working Conference on Reverse Engineering*, Toronto, Canada, 1995, IEEE CS Press, pp. 237-249

[15] H. M. Sneed, "Object-oriented COBOL recycling", *Proc. of 3rd Working Conference on Reverse Engineering*, Monterey, CA, 1996, IEEE CS Press, pp. 169-178

[16] G.V. Subramaniam, E.J. Bwirne, "Deriving an object model from legacy FORTRAN code", *Proc. of International Conference on Software Maintenance*, Monterey, CA, 1996, IEEE CS Press, pp. 3-12

[17] A.S. Yeh, D.R. Harris, and H.B. Rubenstein, "Recovering abstract data types and object instances from a conventional procedural language", *Proc. of 2nd Working Conference on Reverse Engineering*, Toronto, Canada, 1995, IEEE CS Press, pp. 227-236