A Framework For Reverse Engineering Process

Samuel A. AJILA(Ph.D.)

Department of Mathematics & Computer Science
National University of Lesotho, Roma, 180, Lesotho

e-mail: sajila@macs.nul.ls

Abstract

As the 20th century draws to a close, software has become one of the major driving forces in business, government, military equipment and operations, and in academics. In the business and military sectors, we have millions of lines of old code that need to be “re-engineered” so that they can conform to the new modern technology. One way of getting out useful information from these pile of codes is to apply the technique of reverse engineering. Reverse Engineering also supports comprehension in software maintenance, testing, and quality assurance.

The aim of reverse engineering is to provide comprehensive information about the design concepts included in a program. The idea is to provide abstraction mechanisms to allow an easy understanding of the structure of the program to people without or with a minimal prior knowledge of it.

Reverse engineering from “code to design” can be carried out in two steps. First, the source code need to be analyzed detecting all the structure that may contain information for the design. Then, with this “skeletal” information from the code, the user has to provide semantic information which otherwise is impossible to deduce from the first step.

The information that can be derived from a program depends both on the “programmer” and the characteristics of the source language itself. The choice of the target design method should be based on the ability of the method to represent the derived information from the code and on the knowledge that final users have of the method.

In this paper we propose a framework for Reverse Engineering from Code to Design. This framework is generic and it will be applied to a case study based onAda(a programming language) and HOOD²(a design method). Our choice of Ada and HOOD is based on the tight liaison between both methods. This liaison allows the definition of standard mechanisms to generate Ada templates from HOOD designs.

Keywords: Reverse Engineering, re-engineering, Code, Design, Software, Ada, and HOOD.

---

1 To appear in SACLA ’97, June 23 - 24, 1997, Port Elizabeth, South Africa.
2 Hierarchical Object Oriented Design.
1.0 Introduction

Reverse engineering technique is in general applied to an existing software system. It can be applied to various sub-domains in the software lifecycle. These sub-domains can be categorized into two types: re-documentation and design recovery [Llewellyn 91]. The re-documentation is the creation of one or more representations (data structures, cross references, Entity Relationship diagrams, flowcharts, etc.) at the same level of abstraction. Design recovery is the process whereby the knowledge contained in a particular domain (e.g. code) in a level of abstraction is combined with external supplementary information in order to “re-represent” that same domain in a higher level of abstraction (e.g. design level). In this paper, we are only interested in the design recovery aspect.

Reverse engineering is an abstraction mechanism that provides information about the design concepts hidden in a code. This is to facilitate “easy” understanding of the structure of the program to “people” without or with a minimal knowledge of the code. The reverse engineering technique is useful in the “re-engineering” of old code and generally in software maintenance: reuse, software change, etc. [Ajila 95b].

Reverse engineering rely on being able to recognize, comprehend, and manipulate design decisions in source code. During program development, many decisions are made. Some address the structure of the problem domain and how it should be modeled. Others address functional constraints imposed by the environment, including the implementation language(s) and the target machine. Others are independent of the solution space and they have little effect on the program. The information that can be derived from a program depends largely on the decisions that are made during the program development and the choice of the source language. The choice of the design method should be based on the ability of the method to represent the derived information from the code and on the knowledge that final users have of the method.

The aim of this paper is to present a framework for reverse engineering from code to design. This framework is generic and it will be applied to a case study based on Ada and HOOD. Ada is a good programming language because it contains a rich variety of constructs that can be analyzed automatically and that represent in no little way constructs in other programming languages. HOOD is an Hierarchical Object Oriented Design method. HOOD is one of the software design methods used by European Space Agency for the project COLUMBUS.

The rest of this paper is structured as follows: section 2 defines some of the terms that will be used later in the paper, section 3 gives a generic view on reverse engineering process, section 4 describes the design method HOOD, and section 5 relates the programming language concepts to design decisions. We define our framework in section 6, a case study in section 7, and we give our conclusion and further work in section 8.
2.0 Preliminary definitions

This section summarizes the definitions of some of the terms we are going to use in this paper. We have already defined reverse engineering, re-documentation, and design recovery in section one. The following definitions are based on the work of Chikofsky and Cross [Chikofsky 90]:

- **Forward Engineering.** This is the classical process of developing software starting from the problem definition, then the specification, followed by the design, and eventually the implementation. This is a top-down software development technique based on “Software lifecycle”. The term “forward” is used if it is necessary to distinguished between “forward engineering” and “reverse engineering”.

- **Restructuring.** This is the transformation of a type of representation into another type at the same level of abstraction. During restructuring, the functionality and the dynamic nature of the system is always preserved. An example of application of this technique is the transformation of a “non-structured code” to a “structured code”.

- **Re-engineering.** This is the process of analyzing and modifying a system in order to reconstitute the system in a new form. Re-engineering uses the techniques of reverse engineering and that of forward engineering.

- **Software lifecycle.** Software lifecycle begins with the recognition of some need that leads to the specification, design, and implementation of a software product, and it continues until that product is “retired” [Ajila 96].

3.0 Reverse Engineering Process

According to the report on the “First Working Conference on Reverse Engineering” [Selfridge 94], reverse engineering is an umbrella phrase that covers the general idea of taking an existing software system and extracting from it a “higher-level” description that retains the most salient and important characteristics of the original design. In this section, we present a generic view of reverse engineering process.

The traditional software development technique is based on the software lifecycle which of course is the “forward engineering” approach. The lifecycle approach of systems engineering prescribes a number of phases that should be followed, generally in an interactive and iterative manner, to enable success to be realized in fielding a large-scale system that meets user requirements. Following the requirement phase there is the specification and detailed design phase, this is followed by the implementation phase. The testing is the next phase after implementation and the last phase is maintenance.
Figure 3.1: The Reverse Engineering Process

Figure 3.1 shows an abstract view of reverse engineering process. In this figure we show the contrast between forward and reverse engineering. The forward engineering process is essentially an approach for developing new software product while the reverse engineering process is in general a technique applied on an existing software. The process model is in general the same. The only difference is that one is forward and the other is backward. In this case one can use any of the traditional software development models (e.g. waterfall model).

4.0 Design Process: HOOD design method

Software design is the process of taking a functional specification and a set of nonfunctional constraints and producing a description of an implementation from which source code can be developed [Rugaber 90].

One of the issues (the most important one) that generally arise when thinking about reverse engineering is the form of the “higher-level” representation and the techniques for deriving it. In this paper we are proposing HOOD as a form of representation for the design-level. This form of representation is powerful enough to capture all the design decisions that may be embedded in the code.

HOOD is an object oriented design method. This method makes use of notation close to Ada and it is particularly suitable for designing real time and distributed systems [Lai 91]. HOOD was developed as a result of the industrial experience on Object Oriented Design(OOD) and Abstract Machine(AM). HOOD integrated the concepts of objects and hierarchical structures. It was defined in 1986 by Cisi-Ingenierie(France) in collaboration with MATRA(France) and a Danish company called CRI. HOOD was retained (out of 15 propositions) for the design of the project COLUMBUS by the European Space Agency [Legout 92, Ajila 95].
HOOD provides an object-based notation to describe systems. This notation can be in graphic (cf. figure 4.1) form or in textual form or both. HOOD objects provide an interface to the rest of the system, encapsulate its data and internal structure and may have a particular dynamic behavior. A design step in this method consists of decomposing an object (called parent object) into sub-objects (child objects). HOOD defines two types of objects: active and passive. Figure 4.1 shows the graphical representation and decomposition of objects in HOOD. There are two types of relationships in HOOD: USE and INCLUDE. The USE relationship defines a senior-junior type of hierarchy between objects. The relationships INCLUDE allows a parent object to be decomposed into child objects that collectively offer the same functionality as the parent object. Each of the child object can further be decomposed into sub-objects. This decomposition strategy explains the hierarchical structure of HOOD objects. In spite of the fact that objects can be sub-divided in order to realize a system, recursive nature is not allowed. The textual representation is referred to as Object Description Skeleton (ODS). ODS has a well defined syntax [Lai 91].

5.0 Languages and Design Decisions

Programming languages are always designed to reflect some of the design decisions and to make programming easier. No doubt, there is a correspondence between design decisions and the variety of approaches found in modern programming languages. This done by providing variety of abstraction mechanisms [Rugaber 90]. For example:

- The Algol-like languages: Algol-60, Algol-68, Algol-W, PS-Algol, and Pascal provide mechanisms to support decomposition of procedures into statements and data into its components. This done by the introduction of a systematized data structures.
• Logic programming languages such as Prolog and Highlog favor the function/relation dichotomy. A traditional Algol-like languages programmer may find it difficult to program using Prolog or any other logic-like languages.

• In the same way, programming in functional language may be difficult for a traditional programmer because functional languages do not favor variables instead there is an explicit trade off between data and procedures.

• Generalization and/or specialization are the major issues in most of the object oriented programming languages; e.g. smalltalk, C++, Eiffel, and Java.

• Encapsulation is explicitly put forward in languages like Ada and Modula. The major issue here is that there is a separation between the functional specification of a module from its implementation (i.e. its body). An example of this is seen in Ada package where encapsulation is implemented via information hiding. In Ada package, the functional specification (i.e. facilities provided by the modules) is separated from the definition of what the module is doing (i.e. its body or implementation).

Generally, design involve making a choice between variety of alternatives. However if proper representation is not done, the alternatives considered and most importantly the rational for making the final choice may be lost. It is not enough to simply recognize design decisions in a code. It is equally important to be able to represent this information in a form that will be useful for the purpose of software maintenance or re-engineering. This should be organized in such a way as to make it possible for a programmer or software engineer to use it. This is the reason why we propose a framework for reverse engineering in this paper.

6.0 A Framework for Reverse Engineering: From “Code to Design”

We present below in figure 6.1 a conceptual framework for reverse engineering process. This framework is based on the following:

• The Code is first analyzed in order to bring out the design decisions embedded in the program. This analysis is based on the relationship between the various components of the code. It takes into consideration the syntax and semantics of the programming language used in developing the code. The analyzed information is then stored as a database.

• A set of relations are defined. These relations are based on the dependency relationships between the program constructs. These relations are used to represent the design decisions in the code.
The design decisions are then enriched by providing semantic information which is impossible to deduce from the code.

The enriched design decisions can then be represented in a design method. The choice of the design method should take into consideration the nature of the programming language as well as the abstraction method used in steps 1 to 3 above. In most cases, an object oriented design method will be suitable for object type languages. A logic type representation will be adequate for logic programming languages and we can use any other form of design methods for the conventional procedural languages. The most important thing to note is that the design method must captured the design decisions in such a way as to make it useful later.

A user interface is provided to facilitate the use of the derived information. We also believe that a good method of design must facilitate querying and report generating. This can be done through the user interface.

7. A Case Study

The case study is based on Ada and HOOD. This choice is based on the fact that Ada as a programming language has some constructs that are rich in semantics and can
be defined in two steps: first the declaration and then the body. These constructs are packages, tasks, generic units, private types, incomplete types and deferred constants [Booch 88]. HOOD as a method of design is based on the concept of object and the history of this method is linked with Ada. The translation from HOOD to Ada is possible via the ODS [Legout 92, Lai 91, Ajila 95(b)].

The case study (cf. figure 7.1) consists of: analyzing the code, processing, transformation, and mapping. The static analyzer accept as input the Ada code. It produces a decorated abstract tree of the code. The processor processes the decorated abstract tree and produces data structures representing the relationships and tables containing the various objects and their characteristics (cf. Annex A.3 toA.7). These tables are stored in the database (cf. figure 7.1). The data structure produced by the processor is based on the objects relationships in Ada. The processor makes use of the pre-defined relations in order to produce the data structure. The code transformation picks up the relations(cf. Annex A.8) produced by the processor and transform it into design decisions by mapping HOOD tools and enriching the relations.

Figure 7.1 Architecture of the case study: reverse engineering from Ada to HOOD.

HOOD tools is practically a table of correspondence between Ada objects and HOOD objects (cf. figure 7.2).
The design transformation accept as input the design decisions. These are then transform into HOOD objects and ODS (cf. Annex A.9 and figures A9.1, A9.2, and A9.3). It is important to note here that this is the most difficult part of the case study because we are dealing with both graphic and textual representations. As at present, the graphical part is not yet automated. This aspect demands a graphical environment that can support dynamic graphic development as well as program generation. An example of this type of environment is CONCERTO developed by Sema Group, France [Legout 92, Heitz 92, Ajila 95(b & c)]. In this environment it is possible to move from HOOD to Ada via ODS (i.e. forward engineering) but, the reverse is not yet possible (i.e. reverse engineering).

8. Conclusion and Further Work

One of the aims of reverse engineering is to provide comprehensive information about the design concepts included in a program. This is to enhance re-engineering, reuse, and maintenance of software. Software maintenance is generally considered to be difficult, expensive, and time consuming [Llewellyn 91, Ajila 95(c)]. One of the reasons for this is that it is difficult for software engineers or programmers to fully understand a software system that they did not develop. Even when they develop their own system, it is easy to forget the details of the design or code; more so if there is little or no documentation. It is our believe that reverse engineering tool can fill this gap.

In this paper, we have presented a generic framework for reverse engineering process. This framework can be applied on the basis of the type of application and the design method. We also present a case study based on Ada and HOOD. This case study shows the effectiveness of our framework.

We believe that reverse engineering should be done incrementally. We also believe that reverse engineering process cannot be completely automated [Selfridge 94]. There is the need for the participation of the system engineers or programmers. All that is needed from research point of view is to provide tools and ideas that will make reverse engineering process possible.
As part of further work to be done, we intend to introduce knowledge into the reverse engineering process. We cannot completely represent code in an abstract form. We need to be able to study the structure of programming languages and the dynamics in order to introduce certain form of knowledge into the process. As far as the case study is concerned, the graphical aspect need to be further research so that the participation of the user can be further minimize. There is the need to integrate the case study in a software environment such as CONCERTO [Legout 92] or ALF [Ajila 91].

Finally, we believe that our approach is practical and it will benefit people engaged in software development and maintenance.

References


Annex A

A.1 Problem definition

Imagine that we have a large collection of compact disks. From time to time it possible to add to this collection and to remove those that are outdated. As the number of albums increase, it became very difficult to manage the album base manually. So we decided to construct a database for the albums. We give below the code in Ada written for this system (cf. A.2). This example is taken from the book “Software Engineering with Ada” by G. Booch [Booch 88].

Using our case study prototype (cf. figure 7.1), we are able to extract tables A.3 to A.7. It is important to note that tables A.3 to A.7 are truncated compared to what was generated. This is done in order to reduce the number of pages.

A.2 The Code in Ada

```ada
package project is
    package unit_information is
        type name_type is new string(1..20);
        type programmer_type is new string(1..20);
        type reference;
        type reference_access is access reference;
        type reference is record
            document : string(1..80);
            page : positive;
            next : reference_access;
        end record;
        type status_type is (design, code, test, operational);
        type version_type is range 0..99;
        type unit_record is record
            programmer  : programmer_type;
            status : status_type;
            unit_name : name_type;
            version : version_type;
        end record;
    end unit_information;

    package data_base is
        use unit_information;
        maximum_records : constant := 100;
        type record_index is range 0..maximum_records;
        type project_records is array(record_index) of unit_record;
        active_records : record_index;
        data : project_records;
    end data_base;

    end project;
end project;

with project;
use project;
package inquiry_operations is
    type command is (collect_statistics, list_data, quit, select_unit);
    function request return command;
    procedure collect_statistics;
    procedure list_data;
    procedure select_unit;
end inquiry_operations;

package body project is
    with sequential_io;
    package body data_base is
        package base_io is new sequential_io(unit_record);
        use base_io;
        data_file : base_io.file_type;
        begin
            active_records := 0;
            open(data_file, mode => in_file, name => "project_a/units");
            while not end_of_file(data_file)
                loop
                    active_records := active_records + 1;
                    read(data_file, item => data(index));
                end loop;
            close(data_file);
            begin
                case criteria is
                    when unit => if first.unit_name >
                second.unit_name then return true;
                    else return false;
                end if;
                    when requirements => if
                first.requirements.document > second.requirements.document
                then return true;
                    else return false;
                end if;
                end case;
            end out_of_order;
            begin
                for first_index in 1..size - 1 loop
                    for second_index in (first_index + 1).size loop
                        if out_of_order(records(first_index),
                        records(second_index),
                        criteria) then
```
temp_record := records(first_index);
records(first_index) := records(second_index);
records(second_index) := temp_record;
end if;
end loop;
for index in 1..size loop
list(index) := index;
end loop;
end sort;

with text_io;
use text_io;
package body inquiry_operations is
procedure print(data_record : in unit_information.unit_record) is
package status_io is new
text_io.enumeration_io(unit_information.status_io);
package version_io is new
text_io.integer_io(unit_information.version_io);
use status_io, version_io;
begin
set_col(to => 1);
put(data_record.unit_name, width => 20);
set_col(to => 21);
put(data_record.programmer, width => 20);
set_col(to => 42);
put(data_record.status, width => 11);
set_col(to => 54);
put(data_record.version, width => 5);
set_col(to => 62);
put(data_record.requirements.document, width => 80);
new_line;
end print;

procedure print_heading is
begin
set_col(to => 1);
put("unit_name");
set_col(to => 21);
put("programmer_name");
set_col(to => 42);
put("status");
set_col(to => 54);
put("version");
set_col(to => 62);
put("requirements");
nl;
end print_heading;

function request return command is
user_inquiry : command;
package command_io is new text_io.enumeration_io(command);
use command_io;
type sort_key is (requirement, unit);
sort_criteria : sort_key;
package criteria_io is new text_io.enumeration_io(sort_key);
use criteria_io;
type sort_list is array (data_base.record_index)
of data_base.record_index;
sort_table  : sort_list;
procedure sort (records  : in data_base.project_records;
size     : in data_base.record_index;
criteria : in sort_key;
list     : out sort_list) is separate;
begin
set_col(to => 1);
put("what field do you want to sort on?");
get(sort_criteria);
sort(records  => data_base.data,
size     => data_base.active_records,
criteria => sort_criteria,
list     => sort_table);
put("sorting data...");
new_line;
print_heading;
for index in 1..data_base.total_records loop
print(data_base.data(sort_table(index)));
end loop;
end list_data;

procedure select_unit is
unit_name : unit_information.name_type;
begin
put("what unit do you want information on?");
get(unit_name);
new_line;
put("selecting data by unit name...");
new_line;
print_heading;
for index in 1..data_base.active_records loop
if data_base.data(index).unit_name = unit_name then
put(data_base.data(index)).unit_name = unit_name
end loop;
end select_unit;
A.3 Table of Identifiers

<table>
<thead>
<tr>
<th>No.</th>
<th>Ident_Name</th>
<th>Ident_type</th>
<th>Ident_kind</th>
<th>Line</th>
<th>Ident_parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>project</td>
<td>package_spec</td>
<td>PKS</td>
<td>1</td>
<td>project</td>
</tr>
<tr>
<td>1</td>
<td>unit_information</td>
<td>package_spec</td>
<td>PKS</td>
<td>3</td>
<td>project</td>
</tr>
<tr>
<td>2</td>
<td>name_type</td>
<td>DTD</td>
<td>TYP/NEW</td>
<td>4</td>
<td>unit_information</td>
</tr>
<tr>
<td>3</td>
<td>programmer_type</td>
<td>DTD</td>
<td>TYP/NEW</td>
<td>5</td>
<td>unit_information</td>
</tr>
<tr>
<td>4</td>
<td>reference</td>
<td>incomplete_type</td>
<td>ICD</td>
<td>6</td>
<td>unit_information</td>
</tr>
<tr>
<td>5</td>
<td>reference_access</td>
<td>ACD</td>
<td>ACD</td>
<td>7</td>
<td>unit_information</td>
</tr>
<tr>
<td>6</td>
<td>document</td>
<td>string</td>
<td>RTE</td>
<td>9</td>
<td>unit_information</td>
</tr>
<tr>
<td>7</td>
<td>page</td>
<td>positive</td>
<td>RTE</td>
<td>10</td>
<td>unit_information</td>
</tr>
<tr>
<td>8</td>
<td>next</td>
<td>reference_access</td>
<td>RTE</td>
<td>11</td>
<td>unit_information</td>
</tr>
<tr>
<td>9</td>
<td>status_type</td>
<td>END</td>
<td>END</td>
<td>13</td>
<td>unit_information</td>
</tr>
<tr>
<td>10</td>
<td>version_type</td>
<td>ITD</td>
<td>ITD</td>
<td>14</td>
<td>unit_information</td>
</tr>
<tr>
<td>11</td>
<td>unit_record</td>
<td>RTD</td>
<td>RTD</td>
<td>15</td>
<td>unit_information</td>
</tr>
<tr>
<td>12</td>
<td>programmer</td>
<td>programmer_type</td>
<td>RTE</td>
<td>16</td>
<td>unit_information</td>
</tr>
<tr>
<td>13</td>
<td>status</td>
<td>status_type</td>
<td>RTE</td>
<td>17</td>
<td>unit_information</td>
</tr>
<tr>
<td>14</td>
<td>unit_name</td>
<td>name_type</td>
<td>RTE</td>
<td>18</td>
<td>unit_information</td>
</tr>
<tr>
<td>15</td>
<td>version</td>
<td>version_type</td>
<td>RTE</td>
<td>19</td>
<td>unit_information</td>
</tr>
<tr>
<td>16</td>
<td>data_base</td>
<td>package_spec</td>
<td>PKS</td>
<td>23</td>
<td>project</td>
</tr>
<tr>
<td>17</td>
<td>maximum_records</td>
<td>number</td>
<td>CND</td>
<td>25</td>
<td>data_base</td>
</tr>
<tr>
<td>18</td>
<td>record_index</td>
<td>ITD</td>
<td>ITD</td>
<td>26</td>
<td>data_base</td>
</tr>
<tr>
<td>19</td>
<td>project_records</td>
<td>ATD</td>
<td>ATD</td>
<td>27</td>
<td>data_base</td>
</tr>
<tr>
<td>20</td>
<td>active_records</td>
<td>record_index</td>
<td>OBD</td>
<td>28</td>
<td>data_base</td>
</tr>
<tr>
<td>21</td>
<td>data</td>
<td>project_records</td>
<td>OBD</td>
<td>29</td>
<td>data_base</td>
</tr>
<tr>
<td>22</td>
<td>project</td>
<td>package_body</td>
<td>PKB</td>
<td>34</td>
<td>project</td>
</tr>
<tr>
<td>23</td>
<td>data_base</td>
<td>package_body</td>
<td>PKB</td>
<td>35</td>
<td>project</td>
</tr>
<tr>
<td>24</td>
<td>base_io</td>
<td>package_body</td>
<td>NEW</td>
<td>36</td>
<td>data_base</td>
</tr>
</tbody>
</table>

A.4 Table of Sub-Programs

exit;
end if;
end loop;
end select_unit;
end inquiry_package;

with inquiry_operations, text_io;
use inquiry_operations;
procedure inquiry_about_project_data is
begin
loop
    text_io.put("enter inquiry request:");
    case inquiry_operations.request is
        when collect_statistics =>
            text_io.put("statistics command accepted");
            text_io.new_line;
            inquiry_package.collect_statistics;
        when list_data =>
            text_io.put("list command accepted");
            text_io.new_line;
            inquiry_package.list_data; when quit =>
            exit;
        when select_unit =>
            text_io.put("select command accepted");
            text_io.new_line;
            inquiry_operations.select_unit;
        end case;
    end loop;
end inquiry_about_project_data;

A.5 Table of Sub-Program Parameters

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Para_seq_No</th>
<th>Para.</th>
<th>Para. Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sort</td>
<td>1</td>
<td>project_records</td>
<td>IN</td>
</tr>
<tr>
<td>1</td>
<td>sort</td>
<td>2</td>
<td>records</td>
<td>IN</td>
</tr>
<tr>
<td>2</td>
<td>sort</td>
<td>3</td>
<td>record_index</td>
<td>IN</td>
</tr>
<tr>
<td>3</td>
<td>sort</td>
<td>4</td>
<td>size</td>
<td>IN</td>
</tr>
<tr>
<td>4</td>
<td>sort</td>
<td>5</td>
<td>criteria</td>
<td>IN</td>
</tr>
<tr>
<td>5</td>
<td>sort</td>
<td>6</td>
<td>list</td>
<td>OUT</td>
</tr>
<tr>
<td>6</td>
<td>out_of_order</td>
<td>1</td>
<td>unit_record</td>
<td>IN</td>
</tr>
<tr>
<td>7</td>
<td>out_of_order</td>
<td>2</td>
<td>first</td>
<td>IN</td>
</tr>
<tr>
<td>8</td>
<td>out_of_order</td>
<td>3</td>
<td>unit_record</td>
<td>IN</td>
</tr>
<tr>
<td>9</td>
<td>out_of_order</td>
<td>4</td>
<td>second</td>
<td>IN</td>
</tr>
<tr>
<td>10</td>
<td>out_of_order</td>
<td>5</td>
<td>criteria</td>
<td>IN</td>
</tr>
<tr>
<td>11</td>
<td>print</td>
<td>1</td>
<td>unit_record</td>
<td>IN</td>
</tr>
<tr>
<td>12</td>
<td>print</td>
<td>2</td>
<td>data_record</td>
<td>IN</td>
</tr>
</tbody>
</table>

A.6 Importation Table

<table>
<thead>
<tr>
<th>No</th>
<th>Imported</th>
<th>Importing</th>
<th>Line_No</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unit_information</td>
<td>data_base</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>sequential_io</td>
<td>data_base</td>
<td>33</td>
</tr>
<tr>
<td>2</td>
<td>base_io</td>
<td>data_base</td>
<td>37</td>
</tr>
<tr>
<td>3</td>
<td>project</td>
<td>project</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>project</td>
<td>project</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>text_io</td>
<td>inquiry_operations</td>
<td>107</td>
</tr>
<tr>
<td>6</td>
<td>text_io</td>
<td>inquiry_operations</td>
<td>108</td>
</tr>
<tr>
<td>7</td>
<td>status_io</td>
<td>inquiry_operations</td>
<td>116</td>
</tr>
<tr>
<td>8</td>
<td>status_io</td>
<td>project</td>
<td>158</td>
</tr>
<tr>
<td>9</td>
<td>absolute_io</td>
<td>project</td>
<td>161</td>
</tr>
<tr>
<td>10</td>
<td>relative_io</td>
<td>project</td>
<td>164</td>
</tr>
<tr>
<td>11</td>
<td>criteria_io</td>
<td>project</td>
<td>200</td>
</tr>
<tr>
<td>12</td>
<td>inquiry_operations</td>
<td>project</td>
<td>245</td>
</tr>
<tr>
<td>13</td>
<td>text_io</td>
<td>project</td>
<td>245</td>
</tr>
<tr>
<td>14</td>
<td>inquiry_operations</td>
<td>project</td>
<td>246</td>
</tr>
</tbody>
</table>

A.7 Generic Objects Table

<table>
<thead>
<tr>
<th>No</th>
<th>Generic_Obj</th>
<th>New_Ident.</th>
<th>Line_No</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>base_io</td>
<td>sequential_io</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td>status_io</td>
<td>text_io</td>
<td>112</td>
</tr>
<tr>
<td>2</td>
<td>version_io</td>
<td>text_io</td>
<td>114</td>
</tr>
<tr>
<td>3</td>
<td>command_io</td>
<td>text_io</td>
<td>148</td>
</tr>
<tr>
<td>4</td>
<td>status_io</td>
<td>text_io</td>
<td>156</td>
</tr>
<tr>
<td>5</td>
<td>absolute_io</td>
<td>text_io</td>
<td>160</td>
</tr>
</tbody>
</table>
A.5 The Knowledge Base

array(project_records).
array(sort_list).
array(status_data).
calls(project, print).
calls(project, sort).
constant(maximum_records).
formal_param(criteria).
formal_param(criteria).
formal_param(criteria).
formal_param(list).
formal_param(list).
formal_param(project_records).
formal_param(project_records).
formal_param(record_index).
formal_param(record_index).
imports(data_base, base_io).
imports(data_base, sequential_io).
imports(data_base, unit_information).
imports(inquiry_operations, status_io).
imports(inquiry_operations, text_io).
imports(inquiry_operations, text_io).
imports(project, absolute_io).
imports(project, criteria_io).
imports(project, inquiry_operations).
imports(project, inquiry_operations).
imports(project, relative_io).
imports(project, status_io).
imports(project, text_io).
instantiates(absolute_io, text_io).
instantiates(base_io, sequential_io).
instantiates(command_io, text_io).
instantiates(status_io, text_io).
instantiates(status_io, text_io).
instantiates(version_io, text_io).
is_parameter_of(out_of_order, 1, unit_record).
is_parameter_of(out_of_order, 2, first).
is_parameter_of(out_of_order, 3, unit_record).
is_parameter_of(out_of_order, 4, second).
is_parameter_of(out_of_order, 5, criteria).
is_parameter_of(print, 1, unit_record).
is_parameter_of(print, 2, data_record).
is_parameter_of(sort, 1, project_records).
is_parameter_of(sort, 2, records).
is_parameter_of(sort, 3, record_index).
is_parameter_of(sort, 4, size).
is_parameter_of(sort, 5, criteria).
is_parameter_of(sort, 6, list).
object_declared_in(absolute, collect_statistics).
object_declared_in(absolute_io, collect_statistics).
object_declared_in(active_records, data_base).
object_declared_in(base_io, data_base).
object_declared_in(collect_statistics, inquiry_operations).
object_declared_in(version, unit_information).
object_declared_in(version_io, print).
object_declared_in(version_type, unit_information).
package_body(criteria_io).
package_body(data_base).
package_body(inquiry_operations).
package_body(project).
package_body(relative_io).
package_body(status_io).
package_body(version_io).
package_spec(data_base, 23).
package_spec(inquiry_operations, 54).
package_spec(project, 1).
package_spec(unit_information, 3).
procedure_spec(collect_statistics, 155).
procedure_spec(collect_statistics, 57).
procedure_spec(inquiry_about_project_data, 247).
procedure_spec(list_data, 196).
procedure_spec(list_data, 58).
procedure_spec(print, 111).
procedure_spec(sort, 63).
rec_type_var(document).
rec_type_var(status).
rec_type_var(total).
rec_type_var(unit_name).
rec_type_var(version).
record(reference).
record(unit_record).
record_var(reference, document).
record_var(reference, next).
record_var(reference, page).
record_var(status_record).
record_var(var(unit_record, programmer).
record_var(var(unit_record, status).
record_var(var(unit_record, unit_name).
record_var(var(unit_record, version).
renaming(name_type).
renaming(programmer_type).
uses(active_records, assignment_statement, 186).
uses(active_records, assignment_statement, 40).
uses(active_records, assignment_statement, 44).
uses(active_records, loop_statement, 178).
uses(active_records, loop_statement, 234).
uses(close, function_or_library_call_statement, 47).
uses(collect_statistics, case_statement, 252).
uses(command_io, function_or_library_call_statement, 150).
uses(critical, case_statement, 73).
uses(unit_name, if_statement, 236).
uses(unit_name, if_statement, 74).
uses(unit_name, if_statement, 74).
uses(user_inquiry, function_or_library_call_statement, 150).
uses(user_inquiry, return_statement, 152).
uses(version, function_or_library_call_statement, 125).
uses(width, function_or_library_call_statement, 119).
uses(width, function_or_library_call_statement, 121).
uses(width, function_or_library_call_statement, 123).
uses(width, function_or_library_call_statement, 125).
uses(width, function_or_library_call_statement, 127).
uses(width, function_or_library_call_statement, 187).
variable(absolute).
variable(active_records).
A.9 Design decisions In the Code

From Tables A.3 to A.7 above, we are able to deduce the following design decisions from the code.

We identify the following global entities:

- Global_database_project,
- inquiry_operations, and
- project.

The entity “Global_database_project” is the parent object. Two child objects “inquiry_operations” and “project” are defined in the parent object. The entity “project” implements two operations: unit_information and data_base. Inquiry_operations also implements five operations:

- command_inquiry,
- request_inquiry,
- collect_statistics_inquiry,
- list_data_inquiry, and
- select_unit_inquiry.

With this information about the code and the external semantic information, we can then produce the HOOD representation of the design (cf. figures A8.1, A8.2, and A8.3 below). In this example we did not produce the ODS (Object Description Skeleton) because of space. ODS for each object will take at least a minimum of three pages.
Figure A9.1: HOOD diagram of the object GLOBAL_DATABASE_PROJECT.

Figure A9.2: HOOD diagram for the object PROJECT

Figure A9.3: HOOD diagram for the object INQUIRY_OPERATIONS.