植基於使用案例規格之資料庫應用系統快速發展環境
A Rapid Software Development Environment for Database Applications with Use-Case Based Requirements

紀淵博
Yaun-Bor Jih

朱治平
Chih-Ping Chu

成功大學資訊工程學系
Dept. of Computer Science and Information Engineering
National Cheng Kung University, Tainan, Taiwan 701, R.O.C.
chucp@server2.iie.ncku.edu.tw

摘要

如何降低軟體發展時間及成本一直是研究的課題.其重點是軟體必須以元件爲發展基礎;本文提出一植基於使用案例規格之資料庫應用系統發展環境,可將具使用案例規格之資料庫應用系統自動轉換成程式碼.此環境之架構,組成元件及機制在文內均有介紹.

關鍵字: 軟體快速發展工具,使用案例

Abstract

How to reduce the development time and cost of software has been an important research topic. The general idea to development is that software should be developed on basis of components. This paper proposes a rapid software development environment supporting the development of database applications with use-case based requirements. The components, architecture and working mechanism of this environment are introduced.

Keywords: rapid software development tool, use cases

1. Introduction

Reducing software development cost and time has been an important topic in software engineering. Several current research and development areas - object oriented software development methodology, component-based development model, and rapid application development (RAD) tools, etc. are all aimed at such a target. All of these research or development activities share an idea - software should be developed on the basis of components (or objects) [1,8,12,14].

Database-based applications take a large proportion in commercial software. In the past, database applications developers more or less suffered such experiences as high development cost, hard to maintain, delayed release time, etc. The reasons may account for the used traditional function-driven design methods. Function-driven design often makes a system be composed of function-based modules that are not easily reusable and hard to maintain.

Object-oriented software design has been demonstrated to be superior to function-driven design in some application areas [1] like database management software. In the process of object-oriented software development, object's identification and behavior analysis is a main task. Use-case driven requirement analysis, a requirement analysis technique advocated by Ivar Jacobson at 1967 and has become quite popular recently [9], can be applied to identify objects and their behaviors of an application [8,10].

This paper proposes a rapid database-application development environment that uses a use-case specification language to describe requirements, analyzes this specification and assembles the application under the support of a component library and template programs. The rest of the paper is organized as follows. Section 2 introduces the concepts of use case, a transaction procedure to a function of an application. Section 3 describes the proposed development environment, including the included tools and the application generation mechanism. Section 4 uses an example to illustrate the practicability of the proposed environment. In Section 5 we draw a conclusion and present future work directions.

2. Use case

Use case is an operation procedure conducted by an actor to a system's function. It presents a transaction process between a system's actor and the system. Depending on the application, there may exist many kind, abstract or concrete, of different actors [8].

A use case is a system usage scenario of a specific actor. The scenario can be described by the actor based on his/its usage viewpoints. If we collect all the scenarios, i.e., use cases, and analyze them, we can understand the requirement of the system.

Use-case driven analysis helps to cope with the complexity of requirements analysis process. An analyzer can thus focus on one, narrow aspect of the system usage at a time by independently analyzing different use cases. In Object-Oriented Software Engineering, use-case driven requirement analysis technique can be used to

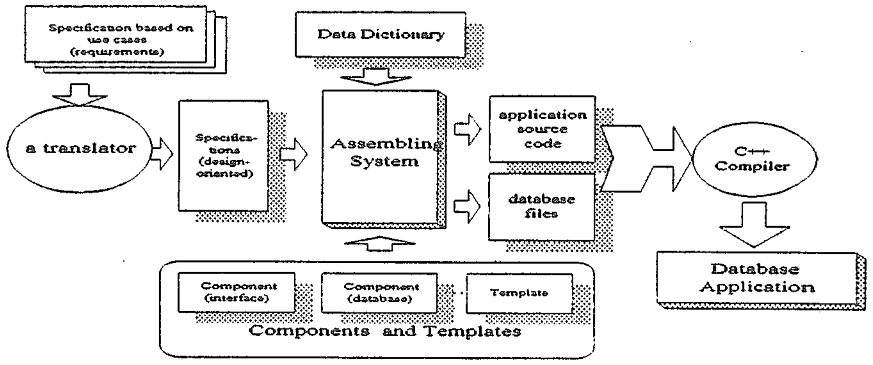

Fig. 1 The system architecture

identify objects and capture requirements in system analysis phase [8,13]. Many existing tools support use-case based software development methodology [6].

Using use-case driven analysis can capture system's requirements, find objects and the inter-object behaviors. But, how to describe those use cases? In general, natural language is used to represent use cases. But there also exist several semi-formal or formal methods to describe use cases [7]. The methods include DFD (Data Flow Diagram), STD (State Transition Diagram) or other graphics tools. However, the graphics methods can not fully describe the details of use cases. Accordingly, those methods must be supplemented by some text description. If we consider simply the expression power of language, we'll find that natural languages is the best choice to describe use case. Other merits existed in natural language include easy-to-write and unnecessary-to-learn. But, there is a serious defect in natural language, i.e., semantics ambiguity.

3. Rapid database-application development environment

The rapid application development environment proposed in this section can help programmer develop a database-related application executed in Microsoft Windows environment promptly. The following tools are included in the environment:

1. A formal use case specification language for describing each use case of the application.
2. An intermediate, design-oriented language for depicting user-interfaces and lower-level execution flows of the application that is transformed from the initial use-case specification.
3. A data dictionary specification for depicting all the data and their attributes accessed in the application.
4. A component library for saving components (objects) that are required and reusable in the application.
5. A set of template programs with assembling directives for guiding the composition of compound window components, sub-application components and the application.
6. A translator and an assembling system for converting a use-case based requirement specification into the source code of the application.

Currently, this environment is built based on Borland C++ programming environment. Fig. 1 shows the system architecture of the environment. In the following subsections we'll explain the implementation details and working mechanism of these tools.

3.1 A use case specification language (UCSL)

As stated before, natural language and other semi-formal, formal tools do have undesired shortcomings when used to describe use cases. To improve such a defect, a formal use case specification language UCSL that eliminates the ambiguity of semantics of natural language was designed. The purpose of using UCSL instead of natural language to describe use cases is for capturing the application requirement explicitly and concisely, such that the subsequent transformations and generation of application can be processed mechanically.

We first define the symbols used in UCSL and other related specifications below.

Symbol	Meaning
Bold-faced lowercase	Reserved word.
[X|Y|..]	Brackets enclose optional items X, Y, .. separated by a vertical bar; one of them must be chosen.
[[..]]	Double brackets indicate enclosed contents can appear zero or more times.
[[X|Y|..]]	The action to [X|Y|..] can be repeated zero or more times.
{}	Curly braces indicate enclosed contents can appear one or more times.
()	Parentheses indicate enclosed contents can appear zero or one time.
< >	Inside < > is listed user-defined term.

The syntax of UCSL is presented below, while the semantics and grammar are shown in [10]. To make the succeeding transformation (explained later) easy, a few statements with design features are also included in UCSL.

project <project_name> **with**

```
caption<project_caption>;
  actor is <actor_name>;
  usecase body
<system_name>.<task_name>.<usecase_name> is
  begin
      select system <system_name> with caption
<"system_caption">;
      display tasks of <system_name>;
      select task <task_name> with caption
<"task_caption">;
      display transactions of <task_name>;
      select transaction <usecase_name> with caption
<"usecase_caption">;
      display blank record of <task_name>
                from category
<dictionary_name>.<database_name>.<table_name>
                [[ and
<dictionary_name>.<database_name>.<table_name> ]]
                where record is <attribute_name>
[[ and <attribute_name> ]];
    fill [record of <table_name>
          | { record of
<table_name>.<attribute_name> (format
<"input_format">)}] ;
      [[ choose record of
<table_name>.<attribute_name>
          either <"caption_name1"> or
<"caption_name2"> ;]]
      [ insert display record of <table_name>
(according to <expression>);
      | delete display record of <table_name>
(according to <expression>);
      | modify display record of <table_name>
(according to <expression>);
      | query display record of <table_name> match
<attribute_name>;
      | browse record of <table_name> list order as
<attribute_name>
        [[ and <attribute_name> ]] ;
      | print a report format :
          title is <"title_description">;
          header is <"header_description"> with
          [table of <table_name> , attribute of
    <attribute_name>|blank]
              ,length <constant>
          [[ and [table of <table_name> , attribute of
    <attribute_name> | blank]
              ,length <constant>]] ;
          (footer with
              [table of <table_name> , attribute of
          <attribute_name>|blank]
                  ,length <constant>
              [[ and [table of <table_name> ,
          attribute of <attribute_name> | blank]
                  ,length <constant>]] ; )
    | compound_scenario ]
    click button of <"button_caption"> ( prompt is
<"error_message">);
        ( catch error message is <"error_message">; )
    end;
  compound_scenario :
      [[ if <expression> compound_scenario endif ;
      | if <expression> compound_scenario1 else
compound_scenario2;
      | while <expression> do compound_scenario done
      | repeat <constant> do compound_scenario done
      | fill one blanked record of <table_name>;
      | change record value of <table_name> with
        [<expression>|<arithmetic_expression>|<field_na
me>]
        ( according to <expression>) ;
      | delete record of <table_name> ( according to
              <expression>) ;
      | search record of <table_name> match
<expression>;
      | skip <constant> record of <table_name>;
      | top <table_name> (according to <expression>);
      | bottom <table_name> (according to
<expression>);
      | move <table_name> to <table_name> (according
to <expression> );
      | calculate <field_name> all value to
[<field_name>|<variable> ]
            [[ depend on [<field_name>|<expression>]
            [[ , [<field_name>|<expression>] ]] ]];
      | prompt is <"prompt_string">;
      | catch error message is <"error_message">;
      | break scenario; ]]
```

Undoubtedly, compared with natural language, UCSL has limited expression power. However, in domain-specific applications this problem may not be that serious if UCSL is provided with sufficient but limited lexemes.

3.2 An intermediate language - GOWL

Since UCSL is mainly used to describe the requirements, it pays only a few concerns on design description of the application. Undoubtedly, it is quite hard to translate a UCSL specification into its corresponding application's source code (C++). There exists a big gap between these two languages. Therefore, to reduce the complexity of the assembling system and make the whole development environment maintainable, providing an intermediate language that is used to describe design details of the application is desired. The one provided in the environment is named GOWL because it is for generating Borland OWL (Object Window Library) source codes.

GOWL is a lower-level (compared with UCSL), abstract formal language that describes the application's I/O interfaces and execution flows. GOWL is mainly used to express another form, that is transformed from UCSL, of the application.

In addition, we also need to define a data dictionary specification. The data dictionary is used to specify the data and their attributes used in the UCSL specification language and GOWL intermediate language. The data dictionary specification can be used to generate database files (in the proposed environment it is .DBF file) and offer much information required in the process of assembling application. We first define the specification of data dictionary and then we'll present the syntax of GOWL.

3.2.1 The data dictionary specification

The data dictionary specification needs also to include the names of database files and tables, in addition to the names of fields, the attributes and the type of attributes. We present the specification of data dictionary below. An explanation to the semantics and the grammar is shown in [10].

```
        dictionary <dictionary_name>
              {database <database_name>
              {table <table_name>
                begin
{<field_name>,<"description">,<type>(,<width>(,<dec
>) )} end }
             end_database }
        end_dictionary
```

3.2.2 The syntax of GOWL

The GOWL specification plays an important role to the final assembling process. Its structure includes two parts, one is project's definitions, defining the menu items of all sub-systems; another is the description of execution flows of all subsystems which contains the description of user I/O interface controls and database file manipulations. The syntax of GOWL is listed as follows. The semantics and grammar of GOWL are shown in [10].

```
project <project_name>
mainwindow <project_name> <"project_caption">
   menu
      [[ popup <"popup_caption">
      {menuitem <"menu_item_description"> do
<system_name>}
      end_popup ]]
   end_menu
end_project

[[system <system_name>
dictionary <dictionary_name>
table <database_name>.<table_name>
[[,<database_name>.<table_name>]]
primary : <field_name>      [[,<field_name>]]
[[ foreign : <field_name>[[,<field_name>]] ]]
[[ set relation from <field_name> to <field_name>]]
  window
       [[ display <"display_caption"> <field_name>
       [[,<"display_caption"> <field_name>]] ]]
       [[ logic_opt <field_name> <"display_caption">
          value [<constant>|<"string">]
          [[,<"display_caption"> value
[<constant>|<"string">] ]]
          default <"display_caption">]]
       [[ button
<"button_caption">[[,<"button_caption">]] ]]
       [[ input <field_name>
          ( format <"input_format">)( check_id )
          [[, <field_name> ( format <"input_format"> )
( check_id ) ]] ]]
       [[ click <"button_caption">
       [ do [ simple_statement | compound_statement |
<system_name> ] done ] ]]
       end_window
       end_system]]

simple_statement:
       [ db_insert ( where <expression>)
       | db_update ( where <expression>)
       | db_delete ( where <expression>)
       | db_query match [<field_name>|<expression>]
       | db_browse <table_name> sort <attribute_name>
        [[+ <attribute_name>]]
       | print using <"print_format_file_name">
        ( where <expression>) ]
       ( prompt <"prompt_string">)
       ( exception <"exception_message">)

compound_statement:
        [[ if <expression> compound_statement endif
         | if <expression> compound_statement else
compound_statement endif
         | while <expression>do compound_statement
done
         | repeat <constant> do compound_statement
done
         | new_record <table_name>
         | replace [ all ]
              <field_name> with
              [<arithmetic_expression>|<field_name>]
              [[,<field_name> with
              [<arithmetic_expression>|<field_name>] ]]
              ( where <expression>)
         | delete <table_name> (where <expression> )
         | search <table_name> match <expression>
         | skip <table_name> <constant>
         | top <table_name> (where <expression>)
         | bottom <table_name> (where <expression>)
         | move <table_name> to <table_name> (where
<expression> )
         | amount <field_name1> to
[<field_name2>|<variable> ]
           [[ depend on [<field_name>|<expression>]
              [[ , [<field_name>|<expression>] ]] ]]
         | prompt <"prompt_string">
         | exception <"exception_message">
         | break ]]
```

Since the database application usually contains printing functions to print various format reports, so the report format specification is also included in GOWL.

```
title <"title_description">
  [date_pageno | pageno_date ]
end_title
header <"header_description">
detail
  { [<field_name>,<print_length>|
space,<space_length>] }
end_detail
(footer
  { [<field_name>,<print_length>|
space,<space_length>] }
end_footer)
```

3.2.3 The applicability of GOWL language

In general, a database-based application includes two kinds of operation: simple data management and complicated data processing.

The simple data management includes five major functions: insertion, modification, deletion, searching and browsing. These functions can be described directly by using the GOWL instructions: db_insert, db_modify, db_delete, db_search and db_browse.

The complicated functions occurred in some applications can be described by using the compoundstatement of GOWL. The compound_statement varies with that of dBASEIII language. dBASEIII's instructions include interface I/O and data manipulation. In interface I/O, the corresponding GOWL interface instructions are provided. In data manipulation, all instructions regarding with or without single record operations can be expressed by the GOWL's statements. In addition, the program control flow statements of the third generation language such as if-, while- and repeat- statements are also included in GOWL to enhance the semantics expression power. In Table 1 we list the corresponding statements between dBASEIII and GOWL.

dBASEIII	GOWL
APPEND [BLANK]	new_record <table_name>
COUNT [<scope>] [FOR/WHILE <condition>] [TO <memvar>]	if <expression> amount <field_name1> to [<field_name2>|<variable>] endif
DO CASE...ENDCASE	if ... else if ... endif endif
DO WHILE ... ENDDO	while ...do ...done
FIND <character string>	search <table_name> match <expression>
GO/GOTO <expN>/Bottom/Top	skip <table_name> <constant>
IF ...ELSE...ENDIF	if ... else ... endif
REPLACE (<scope>) <field> WITH <exp>[[,<field2> WITH <exp2>]]...(FOR/WHILE<condition>)	replace (all) <field_name> with [<exp>|<field_other>] [[,<field2> with[<exp2>|<field_other>]]](where<expression>)
SKIP (<expN>)	skip <table_name> <constant>
SUM (<scope>) (<expN_list>)(FOR/WHILE <condition>)(TO <memvar list>)	if <expression> amount <field_name1> to [<field_name2>|<variable>] endif
TOTAL ON <key field> TO <file name> (<scope>)(FIELDS <field list>) (FOR/WHILE <condition>)	amount <field_name1> to [<field_name2>|<variable>] [[depend on [<field_name>|<expression>] [[, [<field_name>|<expression>]]]]]

Table 1. Statements mapping between dBASEIII and GOWL

3.2.4 The UCSL-to-GOWL translator

In OOSE, the use case's description can help one to identify objects from the description. However, there possibly exist some common objects among use cases [8]. Therefore, how to merge same objects or distinguish them is a problem the designer must handle. As there possibly exist common objects among use cases, in general, we should use an approach shown in [7,8] to check the common objects and functions.

In the proposed environment we adopted a simple method to distinguish those objects with same names. The method is to check the user-defined names including <system_name>, <task_name>, and <usecase_name> in UCSL. If the three names are same, their functions will be merged. If anyone is different, another subsystem or functions will be created.

Each use case description is usually saved in a file. After we define all use cases, we must use a project file to save their file names. So, the UCSL-to-GOWL translator's task is to read a project file, input all the use case files, capture all information in the use cases and store them into a pre-defined internal data structure, and finally use the information obtained to generate GOWL specification file.

3.3 The components library

Three kinds of component are observed in database-based applications. They are user-interface (i.e., window) components, database components and sub-application components. User-interface component can be classified into basic window component and compound

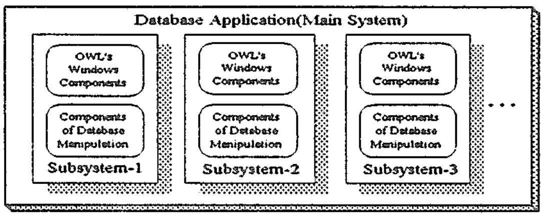
Database Application(Main System)
OWL's Windows Components
Components of Database Manipulation
Subsystem-1
OWL's Windows Components
Components of Database Manipulation
Subsystem-2
OWL's Windows Components
Components of Database Manipulation
Subsystem-3
...

Fig. 2 The architecture diagram of components

window component (i.e., a window component contains another simple window components). Fig. 2 shows the relationship of these three components in which subsystems 1, 2 and 3 are the sub-application components. The sub-application components are composed of the window components and database components.

The component library only stores basic window components and database components. The compound window components and sub-application components are assembled at the time the application is being generated.

3.3.1 User-interface (window) components

We choose Borland Co.'s products, OWL (Object Windows Library), as the source of basic user-interface (window) components. The reasons and the various classes hierarchical diagram of window objects are given in [3,4,10]. The compound window components will be assembled by basic window components based on GOWL specification at application-assembling stage.

3.3.2 Database components

Components (objects) for manipulating database in MS Windows environment have been developed. There is an abstract database class, DATABASE, with those virtual member functions that are for basic manipulations of database file (inserting, modifying, deleting, ..., etc.). All various database classes (e.g., DBASE, Oracle, Informix, Sybase) can inherit from it and override its member functions to achieve various database's functions. We have developed the dBASEIII-compatible file format. So, we have a class named dBASE that inherits from the DATABASE class. We use this class to generate the components for database operations in various applications. Table 2 shows partial member functions provided in our dBASE class.

3.4 The assembling mechanism

In this section, we present the principal technique of the proposed environment, the assembling mechanism. The

The Member Functions of Database Class	Meaning
int dBASE::use(char *fn);	Open .DBF file, return 1 if succeed, otherwise return 0.
int dBASE::read();	Read current record, return 1 if succeed, otherwise 0.
int dBASE::ReadInput(char *fdna,Tedit *tedit, int leng);	Read Window's input control component, tedit; read a string with length leng and field name fdna. Return 1, if succeed, otherwise return 0.
int dBASE::REPLACE (char fdna,Tedit*tedit,int leng);	Read Window's input control component, tedit, read a string with length leng, and modify the field, named fdna. Return 1if succeed, otherwise return 0.
dsvoid dBASE::ShowValue(char *fdna, Tedit *tedit,int leng);	Show the data of field named fdna and length leng in Windows' input control component-tedit.
int dBASE::del();	Delete the current record.
int dBASE::search(char *fdna,char *s);	Search the current record with field name fdna and string's'.
int dBASE::close();	Close file. return 1 if succeed, otherwise return 0.

Table 2. An explanation to partial member functions of the dBASE class

assembling mechanism includes the transformation from Data Dictionary specification to database file and the generation from GOWL to application's source code.

3.4.1 Transformation from data dictionary specification to database files

Transforming data dictionary specification into database files (.DBF files) is simple. We use a scanner, a parser and a semantics analyzer, generated by lex and yacc [11], to do this transformation. If the syntax of the input specification is correct, the .DBF files (corresponding to the tables specification) with the format will be generated by the semantics analyzer. The format of .DBF file includes three areas (Fig. 3). The first area is a header that records the date of file creation, total amount of records, and other information. The second area is an array of field's information, containing the names of attribute of table, the length of each field, and the data type (character, integer, date). The third area is used to store user-input data sequentially. When a new .DBF file is created, only the first and the second areas will be filled with proper data.

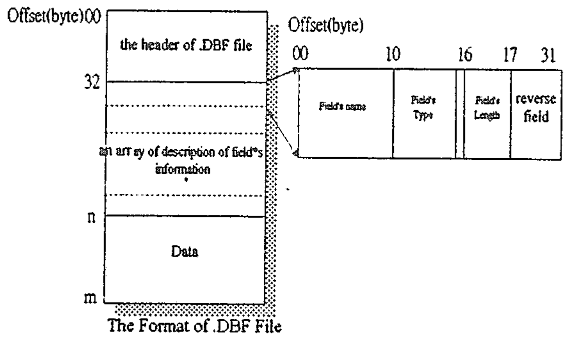

Fig. 3 The format of .dbf file

3.4.2 Generation of the application code

The application source code is actually composed of various sub-application components according to some specific requirement logic. As stated before, the basic window components and database components working for basic operations usually have fixed operation styles and thus are easily reused. These components can be created on basis of the component library and c++ programming [2]. However, the sub-application and compound window components and the application itself generally having complex appearance or flexible operation logic are not easily reused. Therefore, it is needed to use some techniques to construct them. Before generating the application the following two tools and associated data structures must be developed in advance.

1. Use lex and yacc to develop lexical, syntax and semantics analyzers [11] of GOWL specification. The lexical and syntax analyzers are used to check the correctness of GOWL specification, while the semantics analyzer is used to collect the semantics and all related information and stored them in a default data structure. This internal data structure contains an array of pointers that point to different kind of linked list. Each linked list contains the code information.
2. Develop a full-functioned domain-specific database application. Based on the code build a hierarchical template programs (to be explained later) that contain universal (fixed) source codes, including reusable components, associated with assembly-directives. As explained below, the template programs will aid the generation of complex components and application source code significantly.

3.4.2.1 A hierarchical template programs

If one wants to build a complete set of templates of applications, he must first write a complete system code. Those codes must contain all possible subsystem functions. After writing complete source codes, the universal source codes must be identified and extracted as the schema of templates. For those requirement-depended codes, special keywords beginning with a special character such as '$' and with specific semantics are introduced to replace them. The keywords are assembly-directives. In the process of code generation, the assembly-directives can be used to generate different source codes, depending on the requirements of input specification.

To make the composition of application and complex components flexible and the maintenance to the template programs easy, the template programs are organized as a hierarchical organization. That is, based on the program structure the templates are classified into the main program's template, subsystem templates (including sub-subsystem templates), and function templates. Some template programs used in the proposed environment are listed in [10].

The scheme regarding how to use the temple programs to aid the generation of the application and complex components is described in the next section.

3.4.2.2 The assembling process

Fig. 4 presents a general viewpoint to the sub-application components, compound window components and the application assembling process that actually includes two steps. First, use GOWL semantics analyzer to collect all assembly-related information from input GOWL specification and store them in default data structure. After that, read the contents of templates token by token and output the source codes of the sub-application components, compound window components and application either by copying the non-assembly directive code directly or by generating the assembly-directive-induced code according to the information stored in default data structure. Repeat this step until the main program template has been processed.

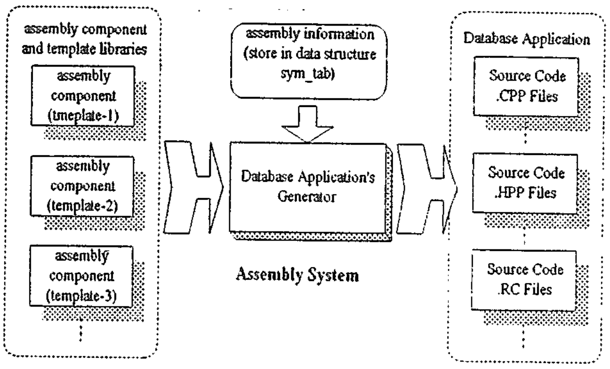
assembly component and template libraries
assembly component (tmeplate-1)
assembly component (template-2)
assembly component (template-3)
assembly information (store in data structure sym_tab)
Database Application's Generator
Assembly System
Database Application
Source Code .CPP Files
Source Code .HPP Files
Source Code .RC Files

Fig. 4. The Assembly Process for Database Application

Let us use an example to explain the assembling process. The file *main.temp* listed below denotes a main program template. At first, the GOWL specification is analyzed. The GOWL semantics analyzer will take the information of assembling obtained in the process of syntax and semantics analyses, store the information into the default data structure (e.g. *sym_tab*), and remark a specific relationship between the information and assembly directives. For example, suppose the assembly directive $OWL_INCLUDE is for generating included OWL files, such as #include <edit.h>, #include <button.h>, etc. and if in the GOWL specification there appear the statements that depict some I/O editing objects and button control objects, the strings, 'edit.h' and 'button.h' must be added into *sym_tab* and such a relationship between these two strings and $OWL_INCLUDE must be marked. Similarly, let $RESOURCE_ID be for generating the declaration statement of resource identifier. If in GOWL specification there is a control component, the semantics analyzer must generate a new identification number for this component,

```
File: main.temp
        #include <owl\framewin.h>
        #include <owl\applicat.h>
        $OWL_INCLUDE
        #include "dbase.hpp"
         #include "bigint.hpp"
        $RESOURCE_ID
        $LOCAL_INC
        ............
        $DEF_RESPONSE_TABLE
        $DEF_SUBWIN
        ..........
```

add the ID into sym_tab, and remark its relationship with $RESOURCE_ID. The same processing mechanism will be applied to handle other directives. After the specification has been analyzed completely, all the information required for assembling is ready. Next, the template *main.temp* is read. When a token is read, it is needed to check if it is an assembly directive. If it is, then read the relational information of assembly from *sym_tab*, assemble those information into C++ statements. Otherwise, copy the original whole statement to the target file. If the meaning of an assembly directive is to process the lower-level template program, like $DEF_SUBWIN, then we must search related template to process it, insert the code at the same position of the target file. Repeat this processing until the last code in main.temp has been processed. Then, all source codes including sub-application components, compound window components and the application will be generated. The source codes obtained will be compiled to get the database's application.

Attributes (color, size, location, etc.) setting for compound window components often brings the programmer difficulty in windows programming. In this environment automatic attributes setting for compound window components is offered. For example, when the system parses the GOWL specification, it will count the number of basic window components and set the layout of compound window components automatically. This will reduce programmer's load significantly.

4. Conclusion and future work

Because of the fashion of object-oriented technique, the advantage of use-case based software engineering has been found recently. This paper described a rapid software development environment, providing UCSL and other input specification languages for generating application promptly. To reduce the complexity of assembling process, an intermediate language GOWL is also designed. The approach of using GOWL and template programs to assemble application is also explained.

In the future, we plan to do the following work:

1. Improve the UCSL and GOWL languages
2. Improve the performance of database class
3. Extend the proposed method to Internet or Web distributed environments

Reference

[1] Booch, G., Object-Oriented Analysis and Design with Applications, The Benjamin/Cummings Publishing Co,Inc.,1994.

[2] Borland C++ Programmer's Guide, Borland Press,1996.

[3] Borland ObjectWindows Reference, Borland Press,1996.

[4] Borland ObjectWindows Programmer's Guide, Borland Press,1996.

[5] Elmasri, R. and Navathe, S.B., *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Co,Inc.,1994.

[6] Harmon, Paul, "Use-Case and OO Analysis," Object-Oriented Strategies, Vol. V, No. 7, pp. 1-6.

[7] Jacobson, I ,"Formalizing use-case modeling," *Journal of Object-Oriented Programming*, June 1995.

[8] Jacobson, I. , Christerson, M., Jonsson P. and Overgaard, G., *Object-Oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley, MA, 1992.

[9] Jacobson, I., Christerson, M. ,"A growing consensus on use cases," *Journal of Object Oriented Programming*, March-April 1995.

[10] Ji, Yaun-Bor, "A Composite Approach for Database-based Applications with Use-Case Based Specifications," Thesis of Master of Science in Information Engineering, National Cheng Kung University, June 1997.

[11] John R. L., Tony M.& Doug B., *Lex & Yacc* , O'Reilly & Associates, Inc.,1992.

[12] Rogerson, Dale, Inside COM, Microsoft Press, 1997.

[13] Rumbaugh, J., "Getting Started - Using use cases to capture requirements," *Journal of Object-Oriented Programming*, September 1994.

[14] Sodhi, J. and Sodhi, P., *Object-Oriented Methods for Software Development*, McGraw-Hill 1996.