CIG: a Class Interface Generator for C++

Ricardo Baeza-Yates *Mario Tichy*

Depto. de Ciencias de la Computación
Universidad de Chile
Casilla 2777, Santiago, Chile
E-mail: {rbaeza,mtichy}@dcc.uchile.cl *

Abstract

Development of an automatic interface generator for C++ classes, with graphical user interface (CIG), is reported. CIG is the outcome of a one semester workshop on object oriented software development for graduate students. The system is written in C++ and C, under the X-Windows environment. This paper gives an overview of the capabilities of the system and its design principles. Concluding remarks include comments on CIG uses and lessons learned from the project as a teaching experience.

Keywords: object oriented programming, C++, class hierarchy, graphical user interface, code generation, X-Windows.

1 Introduction

We feel that the nature of programming is changing. We do no longer think about our programs as individual projects where specific software is produced to meet some particular requirement. As applications become bigger and complex, there is a growing need to build on the work of others as well as on ours. For many years this was attempted through the use of function libraries. As we learned to get profit from the advantages of data abstraction, class libraries began to replace function libraries.

Under the scope of object oriented programming (OOP), software development consists in using objects from existing classes, that send messages one another; and defining new classes as needed.

Class construction involves two stages, that is: class interface definition, and class methods implementation. The later is a creative task that requires knowledge and reasoning. It is hard to automate. The former is a tedious one; often hours of code writing are needed to state a few synthetic ideas. So, it can be easily automated.

CIG was conceived as a programming tool allowing graphical specification of C++ class interfaces using icons, windows, menus, and dialog boxes with mouse support. Afterwards, the system checks consistency on programmer's request and automatically generates code.

We first describe the goals behind the project and the design principles used; then, the architecture of the system is described, and our main results are summarized in the last section.

2 Background

The project had two sets of goals: didactic and applicable. Applicable means that the need for a tool like CIG was rather evident. In the sequel the didactic goals are discussed.

Graduate students in our CS department are allowed to take a course on the theory of OOP. This course is complemented by a one semester workshop on object oriented software development. In

*This work was partially supported by Grant Fondecyt 91-0868 and Grant C-11001 of Fundación Andes.

the workshop, top-down design is taught by walking through designs. Reusability, maintainability and extensibility of programs is tested by trying to apply these concepts to actual code that solve real life problems. The principles of good design and programming are not presented as abstract sermons, but as examples in the context of concrete working programs.

A second concern of the workshop was to teach how to develop software in a team. The project was split into modules and each module was assigned to a group of two students. During the project, the groups were in charge of implementing their module and providing a friendly interface to the remaining parts. Sharing code was steadly encouraged to save time and effort. Extensibility and maintainability of the modules were tested by changes in the design specifications.

3 Design Principles

Design was one of the major challenges proposed in the workshop. A few general guidelines were given as specifications, but everything else was designed by the students. We stressed:

- Design by objectives: before proposing any solution to specific problems or planning any set of tasks, clearly state what are the purposes of the software that is going to be designed. Afterwards, take this in mind along all the design process. Clear objectives avoid wasting time and effort in useless work.
- Top-down design: every one seems to know what this buzz-word means; however, at the workshop we discovered that each one has a slightly different idea about it. Not withstanding this, an effective stratification of the design process was achieved according to different levels of abstraction.
- Object Identification: the question is not "how to find the objects in my problem?", but rather "what are the classes of objects with a common behavior in my problem?". There is no absolute answer which leads to a "software synthesis" technique. A successful selection of base classes requires talent and experience. A general approach and insight, can be found in Meyer [6, Section 14.2, Finding the Classes]. The approach called by Meyer "decomposition evaluation" became particularly fruitful. Since criticism is easier than design, we used a rough decomposition as a starting point. After a few revisions, good designs arisen.
- Building on software tools: great emphasis was put in avoiding to spend time and effort developing what already exists or doing by hand what can be automated. Reading [4] had been strongly recommended, and its philosophy was repeatedly used.

Following the recommendations in [6, Ch. 2] on modularity, CIG was split in a few modules regarding:

- modular decomposability - this is the most intuitive idea about modularity: decomposing a problem into several subproblems;
- modular structure - modules are build to be freely combined for the production new systems, possibly in an environment quite different from the current one;
- modular understandability - each module should be readable and understandable independently from other modules;
- modular continuity - a small change in the problem specification should lead to slight changes in just one or a few modules.

- modular protection - the effect of an abnormal condition in one module during execution, should remain confined to this module, or at least it should propagate only to few other modules.

Finally, it should be pointed that all this fine criteria were applied with a few restrictions regarding real world trade-offs, mainly arising from a limited availability of man-hours. This led, for instance, to grouping together software that otherwise would be split in more than one module, as will be better explained in the following sections.

4 Description of the System

CIG consists in four modules: the user interface, called GUI (for Graphical User Interface); the automatic code generator ACG; the input/output manager IOM; and the syntactic and semantic checker that also performs some additional tasks, arbitrarily included in the same module. Due to time constraints, the later was designated to perform as the communicator among the remaining modules; and, in a first prototype, it even replaced the GUI with a command driven interface. In the workshop, this module was named the Master, and so we will call it here. The interaction among the modules is depicted in Figure 1.

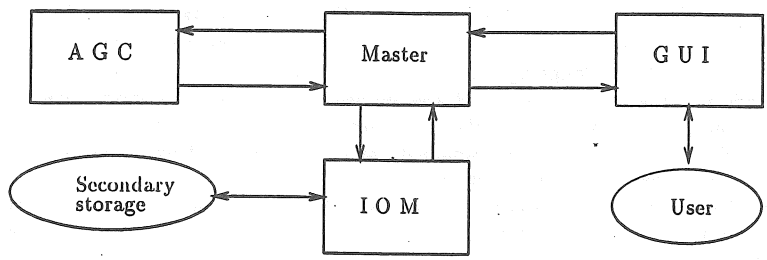

Figure 1: Schematic diagram for the four modules of CIG, and their interaction.

GUI was mainly written in ANSI C [5] and not in C++, to avoid compatibility problems with the code generated by the graphical design tools used under X-Windows. For this task the tool GUIDE (Graphical User Interface Development Environment) of Open Look [7] was used. GUIDE allows to "draw" interactively the interface, generating code that can be compiled under X-Windows using the XView Toolkit and the standard X libraries [8]. GUI is described in section 4.1.

The other three modules were coded in C++ [9], the Master was implemented with the aid of Lex and Yacc [2] (compiler construction tools available in Unix), IOM built on the standard libraries, and AGC was designed and coded as most text processing utilities.

At user's request, GUI tells Master which classes are in the hierarchy to be created, which instance variables they have, which are their methods, and what are the inheritance relationships among the classes. Besides, GUI may ask to perform a consistency checking for the proposed hierarchy, and one or several of the following tasks: storing a hierarchy in a file; deleting a hierarchy stored in a file; retrieving a hierarchy from a file; reading the names of the files in the current directory; changing the current directory; or generating code for a class or a set of classes.

Master passes the information received from GUI to AGC, one class at a time, and the later generates the code in an object of the class "File". This File can be passed to Master that routes it toward IOM which stores it in a class library. IOM is able to store in a file the code generated

within CIG, to retrieve old hierarchies for modifications, to update old hierarchies modified during a session, and to store new hierarchies created during the session. The following subsections describe each module.

4.1 The Graphical Interface

Since CIG was designed to ease the development of class libraries, the effectiveness of its user interface is an essential part of the system. The system provides several services that were assigned to different modules, but the distribution of tasks and the interface for most of these modules is unknown to GUI. The user interface only dialogs with the Master module which is in charge of all the internal interaction of the system.

Regarding the user, it is assumed that he/her is familiar with a window environment with mouse support such as X Windows, MS-Windows or the Macintosh user interface; that knows how to develop C++ class libraries, using conventional programming techniques; and that understands the representation of class hierarchies by means of a directed, acyclic graph. Familiarity with graphical user interfaces developed under Open Look is desirable, but is not a must.

It would be desirable a user interface that prevents or detects errors in the specifications given by the user. However, there are errors and inconsistencies hard to detect. Therefore, additional checkings must be performed by other module in the system. The later should work like no verification is provided by the user interface, and GUI should only take care of the errors that are easy to prevent and that may save time and effort in a later debugging process.

The main objective in designing GUI was to provide the user with a *simple* tool for expressing the structure and the behavior of the classes needed. Simple means a short learning time, with natural and intuitive interaction. Simplicity was seek by reducing the number of options provided. There is an obvious trade-off between simplicity and usefulness; we tried to include every essential option to the user, but not more, sacrificing completeness.

When CIG is invoked, it produces the opening of the main window, which is the parent for all the remaining windows. Edition is controlled by a set of menus available in the main window.

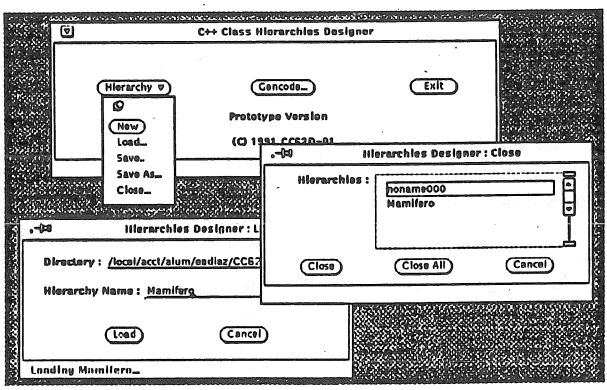

Figure 2: Main window.

4.1.1 The Main Window

This window has a bar with three buttons named (see Figure 2):

- Hierarchies: opens a second menu for hierarchy handling. Its options are:
  - New: opens a new empty edition window to create a new hierarchy.
  - Load: allows to load from a file an already existing hierarchy. The user is prompted to enter the hierarchy's name in a dialog box.
  - Save: allows to save to a file one, several, or all the hierarchies with which the user is working. The details are controlled with the aid of a dialog box.
  - Save as: allows to rename a hierarchy and to create a copy of a hierarchy with a different name.
  - Close: allows to close one, several or all the hierarchies with which the user is working. The names of the hierarchies are selected from a list.
- Gencode: this option gives a directive to generate code for one or several classes of a hierarchy.
- Exit: provides a quick way to close all the windows, saving all the new data and finishing the session.

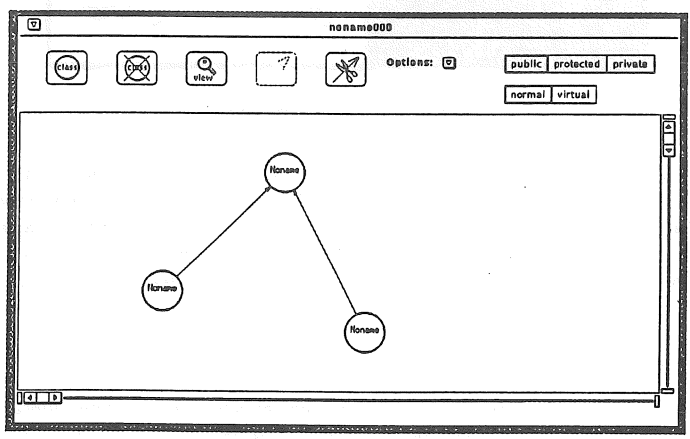

Figure 3: Edition window.

4.1.2 The Edition Window

This window has five icons that represent the available operations for dealing with the classes of a hierarchy. Besides these icons, two sets of exclusive buttons define the inheritance relation between the classes.

The icons (see Figure 3) allow the user to create a new class; destroy a class; open a dialogue box to declare instance variables and methods; create an inheritance arrow to link a pair of classes; and destroy an inheritance arrow.

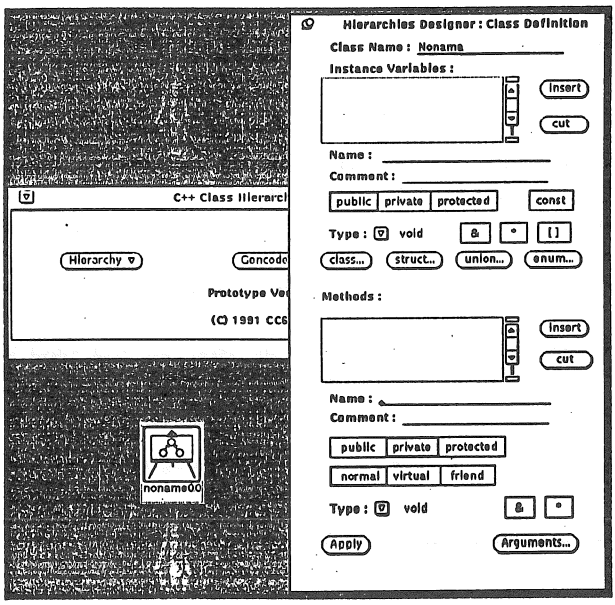

Figure 4: Class information.

A dialog box opens when clicking over the class icon (see Figure 4), where the following information about the class is available: the name of the class (every class, when created, is called "noname"); the names of the instance variables; a comment attached to each variable; the type of the variable; any legal qualifier; the section (public, private, protected) to which the variable belongs; and for each method and their arguments, similar information as that for the instance variables.

4.2 The Master Module

The main purpose of the Master module is to check consistency of user defined hierarchies. Ideally, it should be an object capable of receiving a "check" message and a pointer to a hierarchy, and should return a list of diagnosis messages. However, as already explained, the same module was put in charge of several additional tasks.

4.2.1 Consistency checking

A consistency checker should only verify that the coded information is complete and non contradictory. Some of the consistency rules are already implicit in the user interface. The consistency check includes several aspects, being some of them implemented in the first version of CIG. For example, consistent use of types; name uniqueness; potential problems in multiple inheritance; connectivity of the class hierarchy; etc. Most of the consistency problems are due to some peculiarities of the C++ language which are not usually well known by the user [3].

4.2.2 The additional tasks of "Master"

Besides checking consistency in the specifications given by user, the Master module dialogues with the ACG and IOM modules in order to:

- generate and store code for a class or set of classes;
- manage files and directories in order to store conveniently the information on class hierarchies, without needing to exit the application nor opening an additional window.

At present, GUI creates the remaining three modules of the system as unique objects of their class. In a future version of the system, it is planed to create first the Master, which will create GUI, ACG, and IOM objects besides a separate module exclusively in charge of consistency checking. Under this scheme, it would be possible to create more than a single object of each class (say, several identical modules) for assigning the tasks with more flexibility and running several processes.

4.3 The Automatic Code Generator

The automatic generation of C++ code is done by the ACG module. ACG receives from "Master" the specifications received from the user, one class at a time. This information is structured in such a way that ACG only needs to pass once over the data while it "fills on the blanks" to generate the code. Since CIG is intended only to create the interface for the classes, all the methods are declared "external", or the implementation is substituted by the comment: "Here should be placed the implementation." The user chooses one of these options.

4.4 The Input/Output Module

The interaction of CIG with the file system is done exclusively through IOM, the Input/Output module. IOM dialogues only with "Master", offering the following services:

- Storing all the information concerning one hierarchy in a single physical file.
- Deleting one or several files containing information on hierarchies. The files are selected with the mouse, from the list of files in the current directory.
- Retrieving a file with the information about a hierarchy. The file is selected with the mouse, from the list of files in the current directory.
- Storing the C++ code generated by ACG in a file of a specified directory.
- Listing all the files with information about class hierarchies which belong to the current directory, and listing all the subdirectories in the current directory.
- Warning when a request is going to cause the overwriting of an existing file, and asking if the old file should be copied before saving the new one.

5 Conclusions

The design and implementation of CIG in a workshop on object oriented software development allowed to draw several valuable conclusions from a didactic scope and from the analysis of the system value as a CASE tool.

Looking at the project as a teaching experience, the main conclusions are the following:

- A theoretical course on OOP is not enough to prepare graduate CS students to develop software with this innovative approach. This is true even when the course included homework and small projects on the practical aspects of the subject. We found that important concepts are misunderstood, the value of other concepts is neglected, and several central ideas are rejected due to misconcepts and prejudices.
- Almost all the students that took part in the workshop are not well prepared to work in a team. Usually they fail to explain their design decisions to members of the other groups, they avoid discussing about their own ideas by giving up without arguing or defending their points of view, and they refuse to establish deadlines for their work, rising several bottlenecks during the project.
- During the project, most of the students showed a remarkable positive evolution regarding all of the above mentioned problems. We believe that such courses may significatively contribute to the shaping of well trained engineers that will be able to participate in complex software projects.

Additionally, CIG has attracted the interest of many people involved in software development; it seems to really be a useful tool with a friendly user interface. However, at present, CIG is still a prototype. Some important features are not yet implemented or are implemented in an unsatisfactory form. Finally, the user interface is hardly portable since it was mainly developed with the aid of XView. The project included a Windows 3.0 version of GUI, which is one of the future research lines. The present state of CIG, besides being a very interesting teaching experience, should be regarded as an advanced prototype which may be used to test techniques of stepwise refinement, while the product itself deserves further efforts to reach a production level. For example, to include Booch's notation [1] in the graphical representation.

References

[1] Booch, G. Object Oriented Design with Applications, Benjamming Cummings, 1991.

[2] Brown, D. and Mason, T. Lex and Yacc, O'Reilly, 1990.

[3] Ellis, M. and Stroustrup, B. The Annotated C++ Reference Manual, Addison-Wesley, 1990.

[4] Kernighan, B. and Plauger, B. Software Tools in Pascal, Addison Wesley, 1976.

[5] Kernighan, B. and Ritchie, D. The C Programming Language, ANSI version, Prentice-Hall, 1988.

[6] Meyer, B. Object-oriented Software Construction, Prentice Hall Intl. (U.K.), 1988.

[7] Open Look: Graphical User Interface Application Style Guidelines, Sun Microsystems Inc., Addison-Wesley, 1990.

[8] Scheifler, R., Gettys, J. and Newman, R. The X-Window System, Digital Press, 1988.

[9] Stroustrup, B. The C++ Programming Language, second edition, Addison-Wesley, 1991.