STUDIA UNIV. BABEŞ-BOLYAI, INFORMATICA, Volume XLIII, Number 2, 1998

APPLICATION FRAMEWORK REUSE USING CASE[1] TOOLS

DAN MIRCEA SUCIU

Abstract. The drawbacks of using the generated source code with CASE tools are examined. The major problems found are mainly related to source code generation for architecture classes. This code is incompatible with the application frameworks supported by different development tools. An extension of UML object model with application architecture classes is proposed. Is proved as well that according a special significance to those classes affects the flexibility of generated code and support round-trip engineering.

Key words: object-oriented analysis and design, application framework , round-trip engineering, CASE tools, UML

1. Introduction

The user interface is an important component of an application. However, the design, implementation and test of complex user interfaces are very expensive.

The appearance of visual development tools and environments plays an important role in lighten graphical user interface development. These development environments contain graphical editors, class libraries and/or sets of tools that assist the developer in implementing interfaces functionality. Many of these environments (especially the object-oriented ones) provide generic application architectures and design-patterns which allow developers to focus their attention on specific elements and functionality of applications.

In parallel with development environments, were elaborated object-oriented methods and methodologies for analysis and design of applications and CASE tools that support them. Many of these tools yield substantial improvements in programmer productivity through code generation. Unfortunately, it seems to be many reservations about using the generated code in implementation of final applications. The reservations are mainly induced by the illegibility of generated code and by the conflicts that appear between CASE tools and the *code wizards* provided by development environments. These conflicts are strongly related with application architecture classes and lead to impossibility of using round-trip engineering.

In section two are analyzed these conflicts and the early proposed solutions to solve them. Some of these solutions were applied in different CASE tools that support UML[2] modeling language. Each of these solutions has significant drawbacks, which will be discussed in this section too. Next is proposed an extension of UML language with application architecture classes, which represents the connection between analysis/design models and different application frameworks. In our opinion an

[1] CASE - acronym for Computer Added Software Engineering
[2] UML - acronym for Unified Modeling Language

application framework is characterized by certain classes from class libraries beside tools for automation of implementation (called *code wizards*) provided by development environments.

In the third section is presented the implementation of proposed UML extension in Rocase, a tool for analysis and design of object-oriented applications [CHI97], [SUC96b]. The actual implementation support round-trip engineering using the application framework provided by Microsoft Foundation Class (MFC) class library and Microsoft Visual C++ environment.

The fourth section contains a concise enumeration of main advantages and drawbacks of this proposed solution, and future work proposals.

2. Application framework reuse

2.1. Application architecture classes and frameworks

The graphical operating systems and environments existent on all platforms implies developing of applications with complex graphical user interfaces. The efforts made to realize these interfaces are considerable, even for middle or simple applications.

One of the main goals of visual development environments is to allow application developers to focus on elements related to problem domain. Under operating systems from Microsoft Windows family, environments like Visual Basic, Delphi, Visual C++ or C++ Builder (to enumerate just a few from them) contain resource editors and tools that automate the process of implementation of user interface functionality.

For object-oriented development environments, class libraries play an important role in providing effective *application frameworks*. A framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related applications. Object-oriented application frameworks are a promising technology for reifying proven software designs and implementations in order to reduce the cost and improve the quality of software. According to [FAY97], "a framework is a reusable, semi-complete application that can be specialized to produce custom applications".

In contrast to earlier object-oriented reuse techniques based on class libraries, frameworks are targeted for particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or real-time avionics). Frameworks like MacApp, ET++, Interviews, ACE, Microsoft's MFC and DCOM, JavaSoft's RMI, and implementations of OMG's CORBA play an increasingly important role in contemporary software development.

Application frameworks are composed of general-purpose classes that are intended by their creators to be adapted by others in the future. We will call them *application architecture classes*. These classes define the structure of an application. Subclassing is the usual way that frameworks are meant to be adapted. Typically, the architecture classes are kept pristine, and implementation-specific changes are made to subclasses.

Adaptation by subclassing, however, presents special problems - source code and specialized user knowledge are required. This is the reason for which most of

development environments provide tools that generate code for generic applications and/or aid developers on entire cycle of application implementation. In our opinion, these special tools (called "code wizards") are intimately related with application frameworks.

Many a time, application frameworks are confounded with simple class libraries. The difference between these two concepts is essential. While a class library is composed by classes designed for comprising or deriving into a program, an application framework defines the structure of the program itself. In figure 1 is shown the relationship between these two concepts (note that architecture classes can be included into a more general class library, that contains general-purpose classes as well).

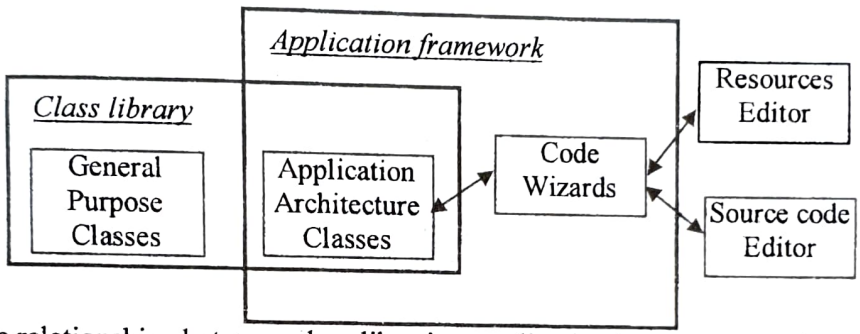

Figure 1. The relationships between class libraries, application frameworks and other components of application development tools

A good example of application framework is, as we said, provided by Microsoft Visual C++ development environment. This framework is composed by application architecture classes from MFC class library and two tools that assist the developer to describe and implement a particular application. These tools (called AppWizard and ClassWizard) together with the resource editor lighten design and implementation efforts for graphical user interfaces. They realize the link between application resources and source code that handle these resources, code that is implemented into application architecture classes.

AppWizard is used just for starting an application development and generates several types of applications, all of which use the application framework in differing ways. *Single Document Interface* (SDI) and *Multiple Document Interface* (MDI) applications make full use of a part of the framework called document/view architecture. Other types of applications, such as *dialog-based* applications, *form-based* applications, and DLLs, use only some of document/view architecture features.

ClassWizard helps developers to create and manage the classes in your program. ClassWizard works both with the classes created initially with AppWizard and the classes created later with ClassWizard. ClassWizard also lets the developers to browse and edit the classes in their program. They can create classes, map messages, override virtual functions, navigate through their application files, and more.

The core of the Microsoft Foundation Class Library is an encapsulation of a large portion of the Windows API in C++ form. Library classes represent windows, dialog boxes, device contexts and other standard Windows items. These classes provide

a convenient C++ member function interface to the structures in Windows that they encapsulate.

Nevertheless, as we showed previously, the Microsoft Foundation Class Library also supplies a layer of additional application functionality built on the C++ encapsulation of the Windows API. This layer is a working application framework for Windows that provides most of the common user interface expected of programs for Windows. Classes in this category contribute to the architecture of a framework application. They supply functionality common to most applications.

In figure 2 are shown the principal architecture classes from MFC together with an example of using them for developing a Multiple Document Interface Pattern application.

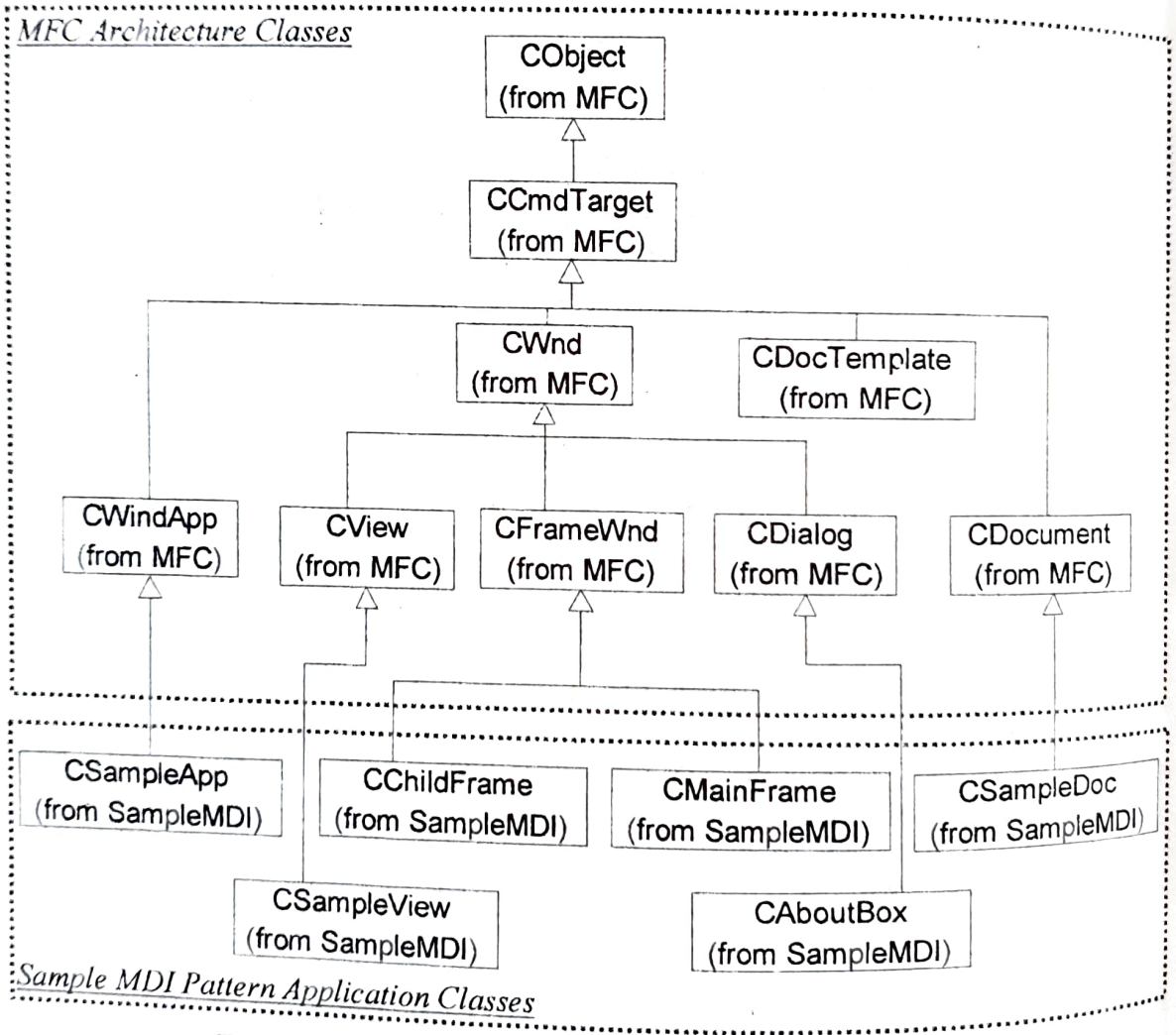

Figure 2. Class hierarchy for MDI applications architecture

All MFC applications have at least two objects: an application object derived from CWinApp, and some sort of main window object, derived (often indirectly) from CWnd (most often, the main window is derived from CFrameWnd, CMDIFrameWnd, or CDialog, all of which are derived from CWnd).

Application architecture classes from MFC model two design patterns: Chain of Responsibility pattern and Model - View - Controller pattern.

The Chain of Responsibility pattern is used for windows messages and user commands routing through objects from application architecture. The classes that model this design pattern are those derived from CCmdTarget. CCmdTarget class from MFC library is the base class that encapsulates code for handling commands and messages and managing message maps. The message maps make the link between windows messages/user commands and member functions implemented to handle them.

CCmdTarget together with its subclasses use a set of C++ *macros* for declaring and defining these message maps. Because there isn't any language mechanisms able to control messages, there are used protocols specific to application framework. ClassWizard helps developers to use this pattern. Nevertheless, this approach supposes insertion of specific comments that limit handling messages source code. It is obvious that this fact induces a particular programming discipline, increases code legibility and eases automatic management of these message maps.

Unfortunately, this specific approach of implementation lead to conflicts when the code generated by a classic CASE tool is used. Therefore, it is impossible to use the both tools (ClassWizard and CASE tool) for automatic implementation of architecture classes. These conflicts are analyzed in detail in next section.

The second design .pattern, Model-View-Controller, is modeled through CDocument and CView classes. This pattern conceptually separates a program's data from the display of that data and from most user interaction with the data. In this pattern, an MFC document object reads and writes data to persistent storage. The document may also provide an interface to the data wherever it resides (such as in a database). A separate view object manages data display, from rendering the data in a window to user selection and editing of data. The view obtains display data from the document and communicates back to the document any data changes.

CDocument and CView are architecture classes. However, they can be exploited into object-oriented models to reuse the design pattern implemented by them. Most of actual CASE tools allow importing of class libraries and reuse of their classes in specific application models. Unfortunately, is difficult to integrate the generated code in final application implementations because CDocument and CView classes are closely related with other architecture classes from MFC (like CWinApp, CDocTemplate, CFrameWnd etc). Classical code-generation mechanisms of contemporary CASE tools don't take in consideration the concepts that lay on basis of MFC applications architecture. Consequently, the developer is constrained to add code manually for linking generated architecture classes. This task is commonly performed by ClassWizard but, as we sow, this tool is helpless to automate it.

This second example of conflict in implementation using both a development environment and a CASE tool have determine us to think more generally at conflicts between these tools. We tried to find a solution that combine both the automation power of application frameworks and modeling power of CASE tools with the methods supported by them.

2.2. *Round trip engineering of application architecture classes*

Code generation represents the ability of a CASE tool to generate code automatically from a design model. In this moment, many CASE tools assist developers on analysis and design phases of applications and allow code generation in a great proportion.

However, developing applications with graphical user interfaces requires resource editors. Although ideal, building a CASE tool that assists developers both in analysis/design stages and in implementation, debugging and testing stages represents a very difficult and laborious task. In addition, the complexity of such tools is very high.

Reverse engineering is another powerful feature of a CASE tool, and represents the ability to create an analysis/design model automatically from source code written into a particular programming language.

Round-trip engineering (or cod-to-model and model-to-code generation) represents the combination of modeling, code generation, coding, and reverse engineering into an iterative cycle. In most cases, this task assumes generation of a particular code, which uses specific language mechanisms for preserving the code added manually between generation and reverse engineering phases. The code that allows this feature is cryptic and not legible.

As we shown previously, the design and implementation of CASE tools that aid developers in all phases of applications life cycle is an expensive task. Consequently, we must focus on optimizing the efforts spent for alternation between model and source code (more precisely, CASE tools and development environments). For this reason, round trip engineering is indispensable in CASE tools functionality. This feature must be flexible so that generated code to be compatible with code wizards from different application frameworks.

In the same vein, another important feature is the quantity of generated code for architecture classes. If a CASE tool don't have any information about application framework structure and architecture classes behavior, the code generated for them is dramatically reduced.

In our opinion, it is very important that a CASE tool to be more related with the application framework. It must know the architecture and "philosophy" of architecture classes from this framework. In the same time, it must play entire or just partial role of code wizards from framework. Of course, these features must not break the generality of the CASE tool or of the method supported by it.

After our knowledge, don't exist an approach that takes in consideration these aspects into a comprehensive way. We can distinguish three approaches of the problems enumerated above, but these approaches can't be considered solutions of them.

The simplest approach, frequently encountered in practice, is to use the code generator of CASE tools just for classes from problems domain, classes that are specific for the developed application. The generated code is then transferred and managed using a development environment. The links with the user interface are implemented exclusive manually and/or using a code wizard. This approach is inefficient even if in analysis/design models are present architecture classes, because these classes are not considered by code generator or reverse engineering tool. Consequently, the code for architecture classes will be written "by hand". For applications with graphical user interface this code used to be massive. As well, this approach can lead to inconsistencies between object-oriented models and source code.

The second approach consists of importing the class library (especially the architecture classes) into a CASE tool project. The imported classes can be derived into analysis/design model to instantiate a particular application architecture. Unfortunately, this is not enough for eliminating problems in transition to programming environments because the code for architecture classes is generated into a "classic" manner. The generated code doesn't take in account the patterns, macros and/or comments needed by environment application frameworks. Therefore, the code wizards become useless. Moreover, the connections between interface resources and generated classes must be done from programming environment, because the CASE tools don't have any information about them.

The third approach supposes generation of generic application using code wizards from programming environments. Then, the generated architecture classes will be integrated into a model made with a CASE tool using reverse engineering. In addition, in this case we will encounter the same problem from above paragraph. If the code generation tool from CASE tool is used, then the framework's specific comments, patterns and macros will be destroyed. The advantage of this approach is that architecture classes can be used into an analysis/design model. However, the source code for these classes must be introduced with programming environment's code wizards.

Although the third approach of implementing architecture classes using CASE tools seems to be more suitable, is far to solve the problems stated in the beginning of this section, because the generation code can be applied only for classes from the problem domain. Nevertheless, the task of implementing architecture classes can also be automated. Consequently, the integration of architecture classes in analysis and design models is just partial.

Therefore, the classical code generation and importing class libraries through reverse engineering are not efficient processes for handling architecture classes from application frameworks.

In our opinion, to solve this problem the CASE tool must be configured in accordance with the framework used in implementation of final application. Our solution implies that the CASE tool can yield some actions proper to framework code wizards. This fact implies that the CASE tool will be "specialized" (it becomes dedicated to one or another programming environment or framework). In practice, this fact doesn't restrict the generality of tool, because the "classical" code generation may be used as well. However, for developers who use the frameworks supported by CASE tool, this "specialization" will be very important.

UML (*Unified Modeling Language*) is the standard object oriented modeling language adopted by the Object Management Group and became the standard support for object-oriented analysis and design. Because UML is the modeling language that has imposed as standard, we will propose an extension of it to allow application architecture classes handling. Therefore, in our extension, the architecture classes are graphical represented in the same manner as ordinarily UML classes with two supplementary optional compartments (figure 3). These compartments show special methods and attributes, those are overriding and/or influence the application framework.

The attributes and methods managed by framework will have specific properties imposed by the framework and its handling features (names, types or

parameters imposed, links with resources, messages or commands handled etc). We will see in section three a particular implementation of these classes into Rocase tool, with respect of Microsoft Visual C++'s application framework features.

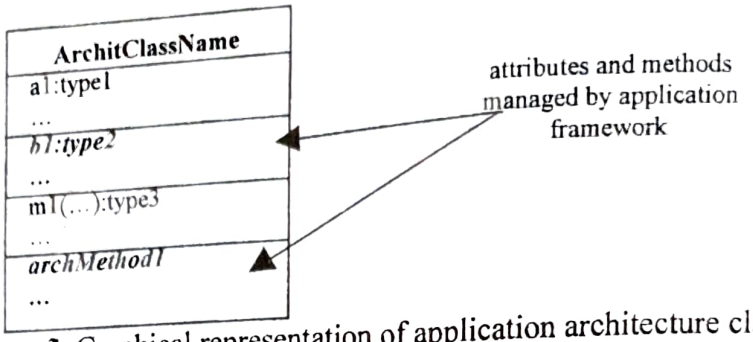

Figure 3. Graphical representation of application architecture classes in our UML extension proposal

From UML models' point of view, the architecture classes have the same behavior like the common classes. There are cases when these classes are not used just for application architecture model. Reuse of design patterns (like Model-View-Controller and Chain of Responsibility patterns from MFC) is just an example of involving architecture classes in problem's domain model.

However, from CASE tools point of view these special classes have a different treatment. The attributes and methods specified in those two new sections will benefit by particular code generation and reverse engineering processes, which must satisfy the characteristics of the framework. In consequence, a CASE tool must contain special modules for source code analysis to perform an exact integration of this code in application framework and to ensure the round-trip engineering accuracy.

3. MFC architecture classes reuse in Rocase

UML is dedicated for a wide range of applications [RAT97]. Object-oriented analysis and design of application using UML is inconceivable in absence of a CASE tool.

Rocase application is a CASE tool that support UML [CHI97]. The extension presented above was implemented in Rocase for Microsoft Visual C++ application framework. The architecture classes exposed in section 2.1 are handled in Rocase so that the generated code respects the ClassWizard's needs from structural point of view.

The associations between Windows messages and classes methods are realized in the same way as ClassWizard do, respecting the same patterns and using identical comments and macros. Practically, when prototypes are implemented, the programming environment is used just for generating a generic application and for resources editing.

In release application implementation phase, when are made code optimizations, Visual C++ programming environment can't be used without any restrictions. Manually added code will be imported through reverse engineering in Rocase projects. This process is realized in two phases: first will be updated the

information about architecture classes, and then a "classic" reverse engineering is used, to update problem domain's classes.

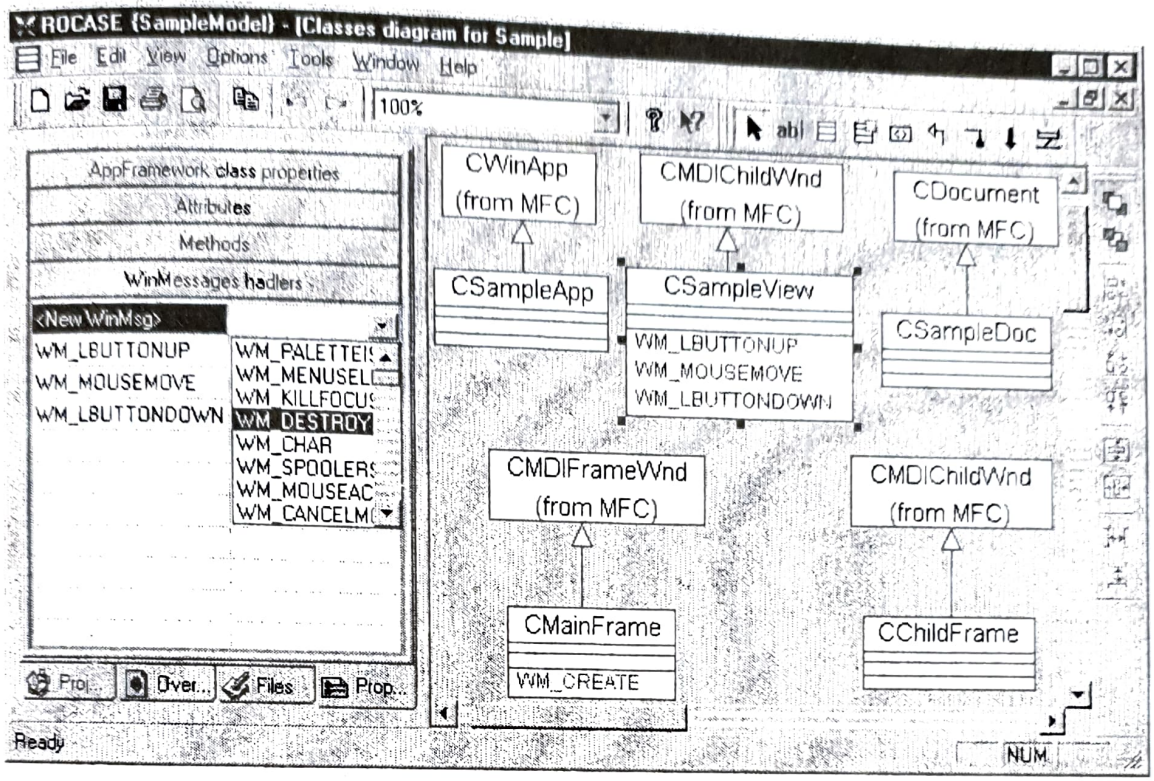

Figure 4. Round-trip engineering of architecture classes in Rocase

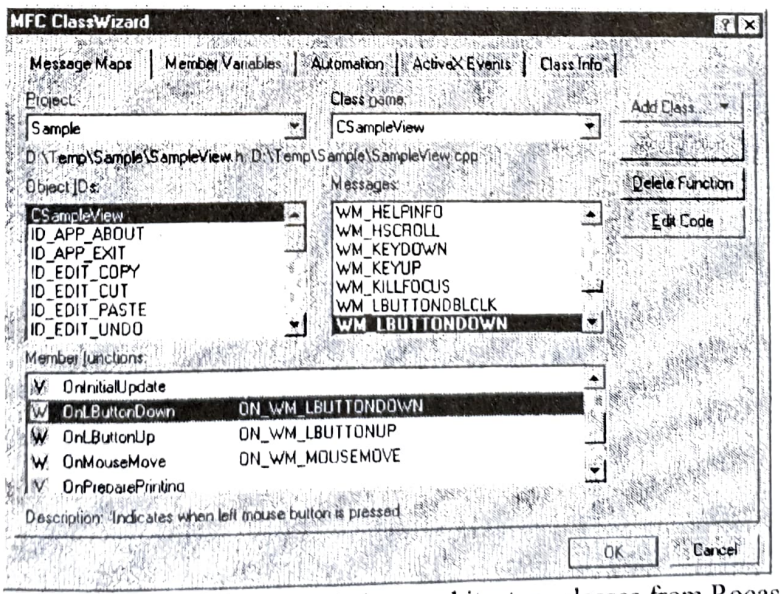

Figure 5. Modifications made into architecture classes from Rocase are visible in ClassWizard window

Moreover, later model modifications (especially addition/deletion of architecture classes) will be reflected in code without affect the code structure. Thus, round-trip engineering if full supported and the stated conflicts are excluded.

Figure 4 shows a sample model realized in Rocase that contains architecture classes. From Rocase was added three methods for class CSampleView that handle WM_LBUTTONUP, WM_MOUSEMOVE and WM_LBUTTONDOWN Windows messages. The generated code, presented below, is recognized by ClasssWizard (figure 5). To perform this generation was implemented a special interpreter that parse .H, .CPP and .CLW files. The .CLW file is used by ClassWizard to stock essential information about architecture classes.

```
//{{AFX_MSG(CSampleView)
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
//}}AFX_MSG
```

```
BEGIN_MESSAGE_MAP(CSampleView, CView)
//{{AFX_MSG_MAP(CSampleView)
  ON_WM_LBUTTONUP()
  ON_WM_MOUSEMOVE()
  ON_WM_LBUTTONDOWN()
//}}AFX_MSG_MAP
// Standard printing commands
  ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
  ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
  ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

Figure 6. Declaration of methods that handle three Windows messages. The highlighted lines are generated with Rocase, and they respect the ClassWizard conventions

4. Conclusions

In this paper we analyzed the conflicts arise between CASE tools and programming environments. These conflicts consist in incompatibility of source code generated by CASE tools with code wizards specific to application frameworks provided by programming environments.

At this time there are CASE tools that avoid developer intervention in source code (Rhapsody [HAR97], COVERS [EXP98]). These tools provide compilers and simulators and don't' need a programming environment for manually completion of generated code. Unfortunately, this tools are suited just for prototypes and not for release applications. They don't provide resource editors and/or debugging tools. As a result, when a final application is implemented, the conflicts pointed in our paper persist.

For ameliorating these conflicts, our approach involve an inevitable dependence of CASE tools by programming environments. The dependency of a CASE tool by a certain environment seems not to be a satisfactory solution, because the generality of incipient stage of application development is reduced. Also, is well known that both analysis and design of an application must be accomplished without taking in consideration a particular language or environment.

However, our method has the advantage that the inclusion of architecture classes into a model can be realized in any stage of application life cycle. This method

is very useful for prototyping phase, when the intervention in code is eliminated. When the release application is implemented, the generated code can be improved using the code wizards provided by environment. In this way, round-trip engineering is well supported.

The approach analyzed in this paper was experimentally implemented into Rocase, a CASE tool that support UML, for Microsoft Visual C++'s application framework. At this time in Rocase is implemented just code generation for methods that handle Windows messages. In future, code generators for methods that handle user commands and member variables associated to dialog controls will be implemented.

References

[BOZ94a] Dorel BOZGA, Dan CHIOREAN, Alin FRENTIU, Bogdan RUS, Vasile SCUTURICI, Dan SUCIU, Dan VASILESCU, *OOA&D: the transition among models*, Preprint No. 5, 1994, pp. 37-44

[BOZ94b] Dorel BOZGA, Dan CHIOREAN, Alin FRENTIU, Bogdan RUS, Vasile SCUTURICI, Dan SUCIU, Dan VASILESCU, *RO-CASE - CASE Tool for Object-Oriented Analysis and Design*, Preprint No. 5, 1994, pp. 29-36

[CHI96] Dan CHIOREAN, *Instrumente CASE pentru analiză şi proiectare orientată-obiect*, PC REPORT, 46, 1996, pp. 24-27

[CHI97] Dan CHIOREAN, Iulian OBER, Marian SCUTURICI, Dan SUCIU, *Present and Perspectives in the Object-Oriented Analysis & Design – The RO-CASE Experience*, The Third International Symposium in Economic Informatics, Bucharest, May, 1997, pp. 23 - 29

[EXP98] Experimental Object Technologies, *COVERS - Reference Manual, Release 3.1*, 1998

[FAY97] Mohamed FAYARD, Douglas SCHMIDT, *Object-Oriented Application Frameworks*, guest editorial for the Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997., available on internet at http://www.cs.wustl.edu/~schmidt/CACM-frameworks.html

[HAR97] David HAREL, Eran GERY, *Executable Object Modeling Using Statecharts*, IEEE Computer, 30(7):,pp. 31-43, July, 1997

[ONI96] Fritz ONION, Andrew HARRISON, *Chain of Responsibility and Command Targets*, C++ Report, vol. 8(7), July 1996, pp. 57-63

[RAT97] RATIONAL Software Corporation, *UML Proposal to the Object Management Group, version 1.1, 1 Sept. 1997*, available on Internet at http://www.rational.com/

[RED96] David REDMOND-PYLE, *GUI Design Techniques for Object Oriented Projects*, Object Expert, vol. 2(1), November/December 1996, pp. 24-28

[SUC96a] Dan SUCIU, *O analiză a trei instrumente CASE: S-CASE, Rational ROSE şi Rational CRC*, PC REPORT, July, pp. 32- 35, 1996,

[SUC96b] Dan SUCIU, *RO-CASE – Istoria dezvoltării unui instrument CASE autohton*, PC REPORT, July, pp. 40-41, 1996

[SUC98] Dan SUCIU, Iulian OBER, *Contrucţia sistemelor software - OPEN/OML*, PC REPORT, Feb. 1998, pp. 28-32

Babeş-Bolyai University, Faculty of Mathematics and Informatics, RO 3400 Cluj-Napoca, str. M. Kogalniceanu 1, România.

E-mail address: tzutzu@cs.ubbcluj.ro