Proposal for tools supporting component based programming

Position paper for Fourth International Workshop on Component-Oriented Programming (WCOP'99)

Henrik Lykke Nielsen and René Elmstrom
Danish Institute of Technology
Information Technology
Teknologiparken
DK-8000 Aarhus C
Denmark

Phone: +45 8943 8422
Fax: +45 8943 8710
E-mail: Henrik.Lykke.Nielsen@teknologisk.dk
Rene.Elmstrom@teknologisk.dk

Abstract

Our experiences with basing application architectures on component structures have shown us that component based architectures result in several advantages such as greater reuse and consistency, ease of maintenance etc. But there are also a number of problems arising when using binary components. We see most of these problems falling in one of the following categories: Culture and organisation, documentation and component management. We propose a number of tools and related techniques that will help lessen the problems arising when using binary components. The tools will address issues such as auto-generating documentation, managing components in different configurations and formal specification and run-time test of functionality.

Background

This paper is based on experiences gathered from client driven as well as in-house projects involving in-house developers, client employed developers and developers employed at third-party consulting companies.
The applications being developed are win32 based database applications with GUI front-ends. Our primary development tool has been Visual Basic and the component technology in question is ActiveX.
One project in particular has provided us with intensive practical experience in making component-based development. The application being developed is a typical small-to-medium-sized multi-user application focusing on handling administrative routines as well as providing the basis for a small data-warehouse-like solution. The application is very similar to applications we have developed in the past, and even though the workflow and the specific business-rules are new most of the technical problems are well known and have well known solutions. The project is a client-based project involving participants from third party consulting companies as well as from the client itself. Our own role has been twofold: As architects we have designed the structure of the system, created standards etc, and as component constructors we have created a number of components for implementing solutions and enforcing standards. The decision to base the application structure on binary components has largely been ours and we have primarily used the potential reuse as the primary sales-argument, but we find it just as important that the use of components provides a consistent coding style and encapsulates solutions of technical problems. The division of the involved participants into an architecture & components group and an application development group has been partly inspired by the organisational structure described in [1] and recommended by e.g. Ivar Jacobsen et al in Software Reuse [2]. An inspiration that has been strengthened by the fact that the most experienced developers were placed at a geographical location separate from that of the client.

Motivation

Source-code reuse contra binary components

In our reference project we found several problems in using binary components. Most of the problems had to do with the very essence of binary components.
When we introduced the concept of component based development in the reference project, we insisted (as component developers) to maintain the code of components

developed i.e. we released only the binary versions of the components (and not the source code) to the application developers. This caused quite some resistance among the application developers. They argued that having access only to the binary version of the components would give them a number of disadvantages:

- The source code is the most precise specification of the code's functionality
- If the code does not work we are able to fix the source code version
- If the code needs changed or extended functionality we are able to implement the change if we have access to the source code

We believe that these reasons originate from the attitude that: "We do not trust what we can not verify ourselves".

For an application developer with a deadline the focus on development will usually be on his own project. So if the code he receives from others (be it source code or binary components) has errors or lacks required options he will always consider programming it himself.
For the single developer focusing on his own project the above mentioned reasons seem very valid. Thus the challenge is to address the issues of the developer so that his objections to using binary components will be taken into account but at the same time to comply with the productivity- and quality-related demands of the organisation.

Managing Components
Of the many technical/practical problems arising when developing, maintaining and using ActiveX components the greatest turned out to be the management of the various versions of components and their many interdependencies.

In the following we will propose a number of tools to help solve these problems.

ActiveX components and type libraries

The ActiveX technology includes formal descriptions of component interfaces in the form of type libraries. The tools we propose analyse both type libraries associated with components as well as source code versions of components. The results of such analyses are used for documentation purposes etc.

Given a type library there is a lot of useful information about the structure of the component at the developer's fingertips such as:

Classes: Class name, class type, supporting interfaces
Interfaces: Interface name, members of the interface
Enumerations
Members: member type, member name
Arguments: Argument name, by value/by reference transfer, argument type

For most of the component parts it is possible to define short textual descriptions as well as Help Context ID's linking to an optional help file for the component.

Proposed Tools

Component Browser

By using a component browser it is possible to browse type libraries - and thereby to access the information which is automatically available. Unfortunately neither Visual Basic's Object Browser or Visual Studio's OLE/COM Object Viewer are very well suited for the average application developer's needs. Visual Studios OLE/COM Object Viewer is a very low-level viewer displaying much of the information textually and as such it is a very direct presentation of the type-library. The tool is very useful for developers who are familiar with ODL/IDL, but not for the general Visual Basic developer who simply want easy-to-read and easy-to-understand information on how to use a component. Visual Basic's Object Browser on the other hand is an easy to use tool. But it presents a rather simplified view of a type library, as some information is not accessible in this tool. Furthermore both tools are rather non-hierarchical in their presentation of a component's structure. Most non-trivial components have some kind of inherent hierarchy in their structure. This hierarchy is not directly presented in either of the component browsers. Additionally a data type is often defined as Object or Variant (which both accept any type of object) because there are multiple specific data types which may be used for this value. So only a limited amount of information on object hierarchies can be gained from the static information available in type libraries.

An increasing number of end-user applications as well as system API's expose their interfaces through the use of ActiveX components. This enable an application programmer to interact with existing instances of classes e.g. a document currently loaded in MS Word. As many developers find it easier to understand structures by examining examples than by studying formal and abstract documentation, component hierarchies are often best understood by looking at the contents of such instances.

The first tool we propose is a more advanced component browser than those mentioned above. The component browser should present the structure and contents of type libraries in a flexible way allowing the user to apply his own perspective on the information in the type library.

We would require the component browser to include the following features:

1. Two major views for the user to choose between:
   - A non-hierarchical view as known from the two existing component browsers
   - A hierarchical view presenting the inherent hierarchy in the component in a tree control

   Both major views should be able to use the very literal representation of Visual Studio's OLE/COM Object Viewer as well as Visual Basic's Object Browser's adapted presentation (where default interfaces are automatically combined with the classes etc.).

2. The ability to view different kinds of type libraries:
   - Type libraries referenced from the development tool
   - Type libraries loaded from disk files

- Type libraries reached through instances of component classes

3. Flexible search functions:
   - Search amongst combinations of the above mentioned kinds of type libraries
   - Searching for all available information in the type libraries

4. The option to browse only generic type library information or to include information from instances of the component classes, either by letting the user create instances inside the tool or by letting him programmatically hook up instances upon the component browser.

5. The option to allow the developers to manipulate the class instances by assigning to properties and calling methods through the use of the component browser as part of the debugging and testing tools in the development environment (se [3] for an example of such an environment).

Component Comparison Tool

ActiveX interfaces are identified by GUIDs (Globally Unique Identifiers). If a new interface has a GUID equal to that of an old interface then by definition this indicates that the interfaces are identical. Visual Basic has a mechanism called binary compatibility that eases the development process when creating and distributing components, by ensuring that the component developer does not (by accident anyway) change a given interface.

As a basic principle you should never change an interface of a component. But if the design of components is iterative, each version a refinement of the previous, the first versions of the components will probably be primitive and inadequate. As long as the components have not yet been distributed beyond the scope of the application developers, it may often be simpler and a more pure solution to change the interface of the component as well as the applications using the component instead of accepting badly designed or inadequate interfaces. If however components have already been distributed to and used by a broader audience, extreme care should be taken not to break existing interfaces.

Visual Basic has as mentioned above a built-in facility for checking the binary compatibility of components. But the comparison is rather primitive. Only when backward compatibility will be affected will the facility provide any specific information on the changes to the component in question. This support proved to be inadequate in the reference project. We needed a way to trace the changes to a components interface over time. Because even though Visual Basic allows interfaces to be extended as long as existing clients are not broken, no facility exists in the development environment to compare components and their interfaces, e.g. providing information on added classes, interfaces, members and arguments. A Component Comparison tool that compares components (both compiled components: .tlb/.olb/.dll/.ocx/.exe files – as well as source code versions of components: .vbp files) would be very useful. It would give the component developer invaluable information on the specific changes made to a component during its lifetime.

Information from the Component Comparison tool may be used to automatically create (part of) history/what's changed lists. Such lists have proved to be extremely useful when distributing new versions of components to application developers. Creating the lists automatically would ease the burden when maintaining components.

Documentation Generation
Documentation is a neglected part of many development projects. If the application is well structured and adheres to a consistent coding style and standard the omission of or the placing of low priority on documentation might however not be fatal. A skilled programmer used to the coding style and standard and with experience in the field of the application domain would in most cases be able to follow the flow of a well-structured program without any major effort. But if the same programmer were to use a binary component the situation would prove less simple. As there is no longer access to the source code only syntactical information on the component's interfaces are available (via the type library). Much can be achieved by intuitive and consistent naming strategies for the different elements of the component and by imbedding element-descriptions. But the source code no longer provides the application developer with a formal specification of the component's semantics.

Pre- and post-conditions
As ActiveX components contain no build-in semantic documentation but only a syntactical description of its interfaces, it is in most cases necessary to provide explicit semantic documentation of components. In the reference project we find this to be crucial in order to answer the complaint that binary components as a contrast to source code components do not have their own build-in semantic documentation.
As part of the documentation it would be beneficial to have a formal specification of the components' functionality. This would result in fewer misunderstandings and questionable cases, but it would also enable others than the component developers themselves to verify the functionality of the components.
Using the principles of Design by Contract [4], a first step in formalising the description of the components might be to define pre- and post-conditions for the members of a component.
These pre- and post-conditions should be stored in such a way that they can be automatically included in the generated documentation. We would require these pre- and post-conditions to be independent of the implementation language of a component, as the ActiveX components themselves are language independent. And we prefer not to define the pre- and post-conditions in the source code so that we are able to handle components alike whether or not we have access to the source code or only to the binary versions of the components (e.g. if they were made by third party companies). Finally pre- and post-conditions might be written by designers as part of the specification of components – even before the components themselves were created.
So we suggest that the pre- and post-conditions as well as other extra component information is stored outside the source code.

Run-time component-wrappers
Beside the aspect of documentation, pre- and post-conditions have the advantage of being possible to check at run-time (depending on the way in which the pre- and post-conditions are expressed), and invariants on interfaces can be checked as part of pre- and post-conditions as suggested in [6]. If a component were accompanied by a component wrapper (and maybe matching test-cases) whose purpose would be to check whether the pre- and-post conditions were complied with, the component's users would have more confidence in the component. This would address one of the major problems with binary components contra source code fragments.
Given a component and its pre- and post-conditions it would be simple to generate component wrappers for checking for and logging of violations of pre- and post-conditions. There are several possible ways to express pre- and post-conditions. UML's Object Constraint Language (OCL) has the advantage of standardisation (see a brief introduction in [5]). Another strategy could be to use one of the many existing (scripting) languages such as VBScript, JavaScript (or JScript) etc. Such scripting languages have the advantage of being simple to implement because parsers and interpreters are readily available.

It would be advantageous to have a general system that could easily create wrappers for components. Several architectures for calling such wrappers are possible, e.g.:
1. Change all design-time references to components to point to the wrapper component instead of the real component. Could be a major task if no effective management tool is available. And would be impossible if the clients (which could themselves be components) were only available in binary form.
2. Instead of creating instances directly in clients, object factories could be explicitly called from the clients to return references to either a wrapper object or the corresponding real object. This strategy would allow central configuration of whether to use wrappers or not. Besides a slight overhead in calling the factory this would again be impossible if the clients were only available in binary form.
3. (Extensible) interception (as described in [7]) which makes it possible to call wrappers transparently without changing clients.
4. Manipulation of registry settings so that when clients believe that they call the real component instead they actually call the wrapper.

Only the architectures mentioned in 3. and 4. can make the use of wrappers work transparently.

Several kinds of functionality would be interesting to implement in wrappers, e.g.:

- Pre- and post condition checks
- Performance measuring
- Call statistics: Call frequency and order, logging of arguments etc.
- Run-time visualisation of member-calls
- Logging of errors – possibly as part of a centralised error-handling mechanism

Component management tool
Practical work has shown us that there are multiple and serious problems in the area of component management. Ensuring that different versions of different components enter into the correct combination is basically simple, but due to human errors and pressed deadlines it often tends to fail. In the reference project we have often encountered situations where extensions or bug fixes of components cover more than a single component. We therefore propose a tool that administers components by handling both individual components as well as sets of related components (component packages). The tool should preferably be able to automatically set references to whole packages of components (to various versions of both binary and source code versions of the components), compile components, create dependency files for components and eventually handle registration of components and component packages on application developers' machines.

Conclusion

Even though we find that the most serious problems in committing organisations to component based development is of a cultural and organisational nature, we also believe that progress can be found in additional tool-support. Automated generation of thorough documentation, formal and verifiable specification of functionality and fewer practical problems arising when managing and applying components all further the process of introducing component development in the organisation. The more faith the application developers have in the components they are using the higher is the probability of achieving success.

In the coming months we will implement the tools proposed in this paper and subsequently apply them to selected components used in our development projects.

References

1. Harmon, Paul: Component Methodologies. Component Development Strategies Vol. VIII, No. 11, Cutter Information Group, November 1998, ISSN 1059-4108.

2. Jacobsen, Ivar; Griss, Martin; Jonsson, Patrik: Software Reuse. Addison-Wesley, 1997, ISBN 0-201-92476-5.

3. Kölling, Michael: Teaching Object Orientation with the Blue Environment. The Journal of Object-Oriented Programming Vol. 12, No. 2, SIGS Publications Inc., May 1999, ISSN 1097-1408.

4. Meyer, Bertrand: Object-Oriented Software Construction, Second Edition, Prentice Hall, 1997, ISBN 0-13-629155-4.

5. Warmer, Jos; Kleppe, Anneke: The Constraint Language of the UML. The Journal of Object-Oriented Programming Vol. 12, No. 2, SIGS Publications Inc., May 1999, ISSN 1097-1408.

6. Szyperski, Clemens: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1997, ISBN 0-201-17888-5.

7. Brown, Keith: Building a Lightweight COM Interception Framework Part 1: The Universal Delegator. Microsoft Systems Journal Vol. 14 No. 1, Miller Freeman, January 1999, ISSN 0889-9932.