Component Technology

Wim Groenendaal

Application development is often treated as 'original' rather than 'routine'. If it would be possible to capture and organize solutions we already know, it would increase both quality and productivity. The evolution of the client/server architecture has now reached a new stage: component technology as the ultimate way to promote re-use. If we take a closer look at the "art of component based development" there are a number of different aspects that need attention. This article will address only some of these.

Components

Today, the word 'Component' has an impact, which can be compared with the term 'Object Oriented' a few years ago. Every software vendor has his own 'component based development' or 'component technology' products. Components is the buzzword today, you cannot live without components. But if we talk about components, what do we actually mean by the verb components?

There are many different definitions on components. For this article I will use a rather simple but effective one: *Components are executable pieces of software only accessible through their interfaces.* So in other words tiny applications hold together by middleware.

Whitebox and Blackbox re-use

Re-use is only one aspect in component technology, although not unimportant. Components are not new; in fact the expected benefits of modular programming were already recognized in the early 70's.

This concept of modular programming has evaluated through the years into today's components. Components are in fact nothing more than small applications, performing specific tasks and, as they are isolated, they can be (re)-used in various environments. Although it seems that nothing much has changed over the years there are at least two big differences: Today components are to be distributed (run on different machines) and the same physical component can be used by different logical applications at the same time. Basically component technology is not the only way to achieve re-use. Object orientation promises about the same advantages when it comes to re-use. The difference of component technology compared to object orientation is that components are not a part of the physical application but are autonomous entities. Both component technology and object orientation promote re-use. In the case of object orientation this is model/source re-use which is in fact a form of whitebox re-use. You can actually see/touch the code. Component technology uses blackbox re-use, meaning you cannot see the code but only the functional specification.

Component technology uses blackbox re-use

This element in the component technology may also introduce new phenomenon. As the actual code is hidden, there is no continuous view on the code quality. Now almost any developer is convinced his way of coding is superior. As long as developers

know others can (re)view their code, they will automatically try to produce high quality code. Components are back boxes and one should be aware of the fact that component technology should never be an excuse for poor programming.

The need for an architecture

One way of dealing with programming quality issues is to define a software architecture, which supports component technology. Although component technology can be achieved without the use of object oriented techniques, it can benefit from the advantages of the object orientation world. In fact it is my personal opinion that a good, flexible architecture for component technology is best served with an objected oriented approach.

An architecture does not only bring quality and consistency into an application, it can also provide productivity and something even more important: ease of maintenance. The architecture's ability of being adaptive, extendable and scalable is probably more important than productivity itself.

The virtual system

In order to design flexible architectures it is important to understand the essence of an automated system. It really does not matter at what level; application, component or object, one looks at a system, then basically it comes down to a variant of the same model I call the virtual system. As this article focuses on component technology the remaining part of the article will use this term, but again, the same rules apply to any entity in a system. Any component consists of a number of processes and flows of information between these processes. A process itself has interfaces to the outside (real) world. If one takes a closer look at the type of interfacing it is possible to distinguish four types of interfaces. The user interface, which deals with all interactions with the user including references to the operating system. The internal interface, where the system communicates with other (sub)-systems in the organization such as an accounting system. The external interface, handling requests from/to systems outside the own organization, such as various mail interfaces and finally the data storage interface, providing data manipulation functionality to (relational) databases and flat files.

A component will have at least one of these interfaces and at least one flow of data going in and one going out. There cannot be a component where only data is going in, then in that case we have a so-called 'sink'. A similar rule is valid for the other way around: a process with only data coming out is also illegal. This type is normally referred to as a 'magic bubble'. The presence of interfaces, and we just concluded components cannot live without them, is a crucial factor in the software architecture. From an ideal point of view, a component should be unaware of the physical characteristics of the operational environment, and then changes to associated elements, outside the component, could have influence on the component behavior. This is also known as the 'ripple effect'. By eliminating this 'ripple effect' components become more flexible and maintainable.

Why are there still architectural monsters being developed?

Depending on the nature of the interface (user, internal, external or data) there are various object oriented techniques available to deal with this challenge. Techniques derived from the 'design pattern' literature form the foundation for these kind of flexible implementations. Whatever solution is chosen, it all comes down to one golden rule: interface objects should always handle communications to the outside world. Interface Objects do not only provide a single point of reference to the outside world; they are also capable of connecting otherwise incompatible formats. Software architectures in the format of component frameworks would provide guidance to application developers and would improve the flexibility, quality and re-usability of components.

If we understand this and agree on this, why are there still architectural monsters being developed?

Wim Groenendaal is senior technical consultant in CMG's research center for client/server technology. Through the years he has been involved in the design of software architectures for various software development environments. In one of the current research projects, code name: CODA (Corporate Design Architecture) all these experiences in client/server technology are gathered and (re)-modeled in a CASE tool in order to design a environment neutral client/server architecture based upon the design pattern theory.

Obstacles

Although the importance of a software architecture is generally accepted, there are a number of candidate obstacles. Some of the most important once are:

Skill
developers need to understand the concept of designing architectures.

Priority
The implementation of components becomes more pragmatic as a project deadline starts to loom.

Cost
Architecture is often regarded as pointless luxury once a system is about ready to get shipped.

Perception
Members of a project team have different objectives than system architects.

From experience I found that the best way to deal with these and other related obstacles is to have a specialized team for designing the architecture. This way you have at least the right skills, priority and perception in the same team. What remain are the costs. The design of a flexible software architecture is time consuming and thus expensive. Furthermore there is no obvious direct return on investment. But if one thinks a good architecture is expensive, try building on a bad architecture.

More challenges

Even with a special engineering team, management commitment and budget there is another, often forgotten, element which can easily disturb the strive for re-use. It is simply not enough to develop reusable components if co-developers have no knowledge of the existence of these components. Due to this lack of component management they will just built it again and that is exactly what we do not want to do.

The design of a flexible software architecture is time consuming and thus expensive

---

If re-use is an objective then it is also obvious that there must be an infrastructure to support the process. Too often component management fails as a result of various circumstances. If we take a closer look at the problems that occur when re-use is involved, the major obstacles are the following: accessibility of available components, lack of version control, lack of a proper search engine, and lack of proper usage guidelines. The best way to deal with these circumstances is the use of supporting software, preferable running on the intranet in order to reach all developers in the organization.

What is missing?

I realize that there are much more interesting aspects in using component technology, such as how does one recognizes a component during application analysis. And what about automated testing? Test tools today are designed to test applications, not components. And then the aspect of application maintenance, how do we manage the implementation of new and changed requirements. And last but not least the component operational environment itself with a still continuing battle between the various component models and the stability/functionality of the connecting software: the middleware.

Conclusion

The main focus of this article was about the component itself and I conclude that although component technology does not rely on object orientation it can certainly benefit from the available techniques, especially the theory on design patterns and weak coupling. It is also true, that promoting 'develop for re-use' is a difficult task to accomplish because a developer's intention is to write code and the lack of proper component management. And even then there are a lot of challenges left to overcome before this technology is really mature.