Component Based Development

M. Huizing

The most used buzzword in the IT-world last year, was probably component*, often used as in the combination* component-based development *(CBD). In this article, I will try to illuminate what components are, why it is a buzzword, what we can expect from it, and what is needed to apply components. In this context, the problems arising from the application of components will be mentioned.*

Components and their promise

First, then, what are components? Of course there are a lot of definitions around, but one of the most useful is this: *software components are (binary) units of independent production, acquisition, and deployment that interact to form a functioning system.*

So, typically, components are rather small but independent parts of a system. But a large system as a whole can be seen as a component as well. It is important to recognize components are runtime entities. They exist while the system is running, in fact: the system consists of components, and is a component itself. Components are not just design entities like classes in object-orientation are.

As said earlier, at his moment everybody is raving about components, and seems to expect a lot from it. What is expected form components, and why is everybody that enthusiast? The expected advantages to be derived from the application of components can be summarized as follows.

1. flexibility

Run-time components can work independently, and, if designed properly, are much less dependent on their environment (hardware, system-software, other applications or components). Therefore, component-based systems are much more adaptable and extendable than systems traditionally designed and built. Usually, components are not changed, but replaced. This flexibility is important in two areas:

a. hardware and system software.

Component-based systems are less sensitive to changes in the foundation (for example: the operating system) than traditional systems. This results in a more rapid migration from one operating system to another, or from one DBMS to another. An interesting result is also the possibility of a system in a technically heterogeneous environment.

b. functionality.

Component-based systems are at a functional level much more adaptable and extendable than traditional systems, because most of the new functionality can be reused some way or another or derived from already existing components.

2. reuse

In principle, CBD enables the development of components which completely implement a technical solution or a business aspect. Such components can be used everywhere. Functionality, be it technical or business oriented, has to be developed and implemented just once, instead of several times, as is now typically the case. It will be clear this is a good thing from the point of view of maintainability, robustness and productivity.

Of course, this reuse can be within a company,

but also over several companies. This will be the case of components made by third-parties.

3. maintainability

In a component-based system a piece of functionality ideally is implemented just once. It is self-evident this results in easier maintenance, which leads to lower cost, and a longer life for these systems. In fact, the distinction between maintenance and construction will become very vague, and completely disappear after some time. *New* applications will consist for a very large part of already existing components. Building a system will look more like assembly than really building. Moreover, the large, monolithic systems as we know them, will disappear resulting in a blurring of the borders between the systems.

It is also usual to mention the following points as advantages of CBD:

4. more rapid development/higher productivity of the developers

In principle, CBD will result in a more rapid development of systems, for reasons of reuse among others. In the long run, this higher productivity will be realised. However, in the short run the fruits of reuse will be smaller than the cost of the introduction of a new way of system development. Furthermore, at his moment reusable components are not available in sufficient measure, and on top of that difficult to acquire.

5. distribution

CBD makes it possible to design systems for distribution (on a LAN, or the Internet, or whatever). This is in fact one of the main reasons for the current hype. And of course, it is a huge advantage. However, at this moment distribution usually has a high price in performance and complexity, and a judicious application of distribution is imperative for the time being.

Problems and pitfalls

Of course, this is a rather impressive list of advantages. Are there any drawbacks or difficulties connected to components? As a matter of fact, yes.

To start with, if you want to use components to the best, a proper environment is necessary. Of course, in theory it is possible to build your own environment immediately on top of the operating system. But in practice, this is not possible. Furthermore, one of the most promising aspects of components is reuse, meaning it should be possible to use again a component made in another situation, possibly in another environment. This implies standardization, and in fact this is the most important obstacle to the full blossoming of CBD at this moment. Standards are needed regarding the middleware in which the components are supposed to work. Middleware can mean a lot of things, but what is meant here is: *a communication layer which enables components to send higher-level messages to other components, if needed in a network.* Of course it is open for discussion what belongs to the middleware, and what not. For example: should transaction-handling be part of it? And how about persistence? And so on.

On this field there are now three competing standards: CORBA from the Object Management Group, COM/DCOM from Microsoft and Java's Remote Method Invocation. In fact, these three standards are not completely comparable to each other. CORBA is just a standard, whereas COM/DCOM is not just a standard but also an implementation. CORBA is really language-independent, COM/DCOM supposedly is, and RMI is part of the Java-standard. A comparison between these standards is, though interesting, not in the scope of this article.

At least as important, but less well known are the difficulties connected with designing for component-systems. There are two popular myths here.

CBD and legacy

The first myth is especially popular in large companies with a lot of huge, existing systems, the so called *legacy systems.* These systems often incorporate a lot of functionality, which the company doesn't want to develop again. On the other hand, the existing systems are technically obsolete and often unmaintainable. A possible solution seems to be to divide the legacy system in a couple of *function modules*, to wrap those modules, and to deploy

those modules as components. Technically this is possible, and in the short run, it can be very useful. And again from a technical point of view, these wrapped modules are components. However, the supposition is you get the advantages of components if you call them components. But that is not true. Legacy systems are designed using a development method which is not suitable for the development of components. As a result, those *components* will be much too big to reuse. The inside of the *components* will be the same unmaintainable coding as it used to be. So, although these modules are components in the technical sense, the advantages of CBD will not materialize if this approach is chosen.

The question here is: is it possible to *refactor* existing legacy systems in such a way really useful components will emerge? There is not a definite answer yet, but most experts are pessimistic. This means those legacy systems must probably be replaced somewhere in the future.

CBD and OO

The other myth is: developing components is easy, because components and objects are the same. So just use an object-oriented method, and the components will flourish. Unfortunately, components and objects have a lot in common, but are not the same.

There are at least two challenges concerning a method for CBD. The first one is: how to design components which are as safe, secure and robust as possible, and are still lightweight, and easy to use. The second one is: how to design generic components which are usable in a lot of different environments, and can be made specific for a concrete implementation. Is this to be done with a kind of inheritance? Or delegation? Or by deriving from generic patterns? Or a combination of these?

In the development of a system, a lot of entities are encountered which are in fact a combination of aspects. For example, if a run-of-the-mill order-entry system is developed, something like *order* will spring up. But in fact this simple notion turns out to be a rather complex combination of different aspects: an order can be seen as kind of financial asset. An order must be persistent. An order can only be created by authorized employees, so will have a kind of authorization. An order must be able to *live* on a Unix system and on a NT system. And so on. All these aspects are not specific to order, but generic, and will be relevant in other *entities* as well. These aspects can be seen as orthogonal dimensions. Therefore, they will be implemented in different components, which must be combined one way or another to form the order. How is this to be done? And how are the orthogonal dimensions to be recognized? How can we ensure the dimensions are orthogonal, and remain so? These are some of the questions to be answered by a method for component-based development.

You get the advantages of components if you call them components, but that is not true!

There is at least one element already recognizable in the solution: *patterns.* Patterns are generic solutions to problems occurring in different contexts. They have already proved useful in *normal* object-oriented design, and this kind of abstraction seems very useful to solving the genericity problem in CBD. In a specific situation, you *derive* a component from an already derived and implemented role in a pattern. Components can be coupled to each other in a controllable way. But patterns are not the only buzzword in this context: aspect-oriented programming, adaptive programming, and others are investigated as well.

These issues are especially pressing in the field of ERP (Enterprise Resource Planning)-systems, like Baan and SAP. At this moment, these systems are monolithic, but they need to be broken up in components. This means an ERP system will become a kind of framework in which business component can function. But how the business components are to be recognized or how the ERP-producer can provide generic business components is still in veils. However, it is no wonder companies like Baan and IBM are heavily investing in research and development in this area.

Of course, a lot of people are working on this, so methods for developing components are growing up at his moment. Usually they are based on the

existing object-oriented methods. The UML (Unified Modeling Language) is commonly used, but as a standard the UML just addresses the set of concepts, the diagram technique and so on. The development method itself is not defined by the UML. And object-orientation is a good starting point for a CBD-method, but not sufficient by itself.

In a comparison between object-orientation and component-orientation, what are the similarities and differences? Objects and components are comparable, in the following sense; they are supposed to encapsulate their own state completely, and to provide an interface to the outside world. In the context of object-orientation however, this is true during design, but very often not anymore during the implementation. It is rather common (although regrettable) to haggle with the principles of OO-design in the implementation, for example for reasons of performance. In a component world, interface oriented design and also implementation is crucial, and to some extent enforced by the middleware (CORBA, DCOM, etc). Why? Because it is impossible to predict the possible uses of a component. So in a component, interface design is more important than it usually is in object-oriented design, and encapsulation of the state is enforced. It is not a surprise design techniques like design-by-contract, using pre- and post-conditions are going through a revival as a result of the upsurge of CBD.

The same goes for thread-security and the like: objects are usually not thread-secure, because the designer knows (or thinks he knows) how the objects are going to be used. In a component context he cannot be sure about that, and therefore the components have to be secured.

The main difference between a component and an object is this: *a component is meant to be a runtime entity, whereas an object is an instance of a class.* Objects exist at runtime, but classes are really design entities. Inheritance (a relationship between classes) therefore, is a less useful concept in a component context than it is in an object-oriented context. Because of the possibilities of call backs and the difficulties with ensuring the encapsulation of the state, inheritance is even problematic in a component context. The movement from inheritance based solutions to object composition, message forwarding and delegation, which was already on its way in the OO world, has gained speed in the component world.

Components are deployed independently, and it is impossible to predict how the component is going to be used. As a result, concerns about dependencies between components, call backs, components using each other, but belonging to different threads, safety and security, and so on are much more important than they are in *plain* object-oriented modeling or design. Seen from this perspective, CBD takes the consequences of object-orientation to the runtime environment.

Logistics and organization

There is still another field in which innovation is needed, and which is crucial to the success of CBD. This field is logistics and organization.

Although it is possible to develop and reuse components within just one organization, it is self-evident acquisition from third parties is a very important advantage. But how to acquire these reusable components from the outside? As always, a balance between costs and benefits must be struck. Nobody is investing two weeks time in searching, evaluating and acquiring a component of which the reuse will yield 2 days time. So, if (as is the case nowadays) the searching and so on of a component costs a lot of time (approximately $5-8$ man days for a small component!), only big components will be acquired and reused. It is therefore imperative to shorten the time for searching, evaluating and acquiring components. Only then a real component market will grow up.

Organizations which are supposed to deploy components have now an IT-organization suited to the development and maintenance of large, monolithic systems. Obviously, such an organization is not suited for the development or use of components. Building components (software especially meant for reuse, of which it is impossible to predict how they are going to be reused) demands a more quality-oriented view than *normal* software-development does. And the reuse of components demands a thorough knowledge of how to reuse components, and where to get them. Again a way

of thinking which is not common in the mainstream of software development.

These problems lie a little bit outside the scope most people use when thinking and talking about components. But they are at the core of the question whether CBD will succeed or fail. CBD is not just another technique, it has very strong repercussions on the way of working of people and organizations.

Conclusion

Component-based development has a lot of promises. But it is not a silver bullet. It will not solve all the problems of software development. And a lot of work has to be done before CBD is really possible. Technical standardization is necessary, and a method suited to CBD has to be developed. But perhaps the most immediate bottleneck is the organization and the logistics of CBD. Just using Java or DCOM or a fancy method is not enough; the way IT is organized has to be changed. And only if that happens, will CBD turn not to be yet another fata morgana.

Mat Huizing is working for the ICG Group and located in Cappelle aan de IJssel. He is a specialist in object oriented design and object oriented modelling, with emphasis on component based development, heuristics, and patterns. The ICG Group is a consultancy- and software-house specialized in CBD and OO.
For more information, you can contact Mat Huizing at: mat@icgroup.nl