Component-Based Business
Background Material

On Components

Richard Veryard, October 2000

Summary

- A component is not a fixed thing. It is a declared (and possibly dynamic) relationship between a parcel of capability and a parcel of service.
- The focus is not on the identity of the component, but on the act of componenting: the (always provisional) declaration that a given lump of business capability, or a given lump of software, suitably wrapped, shall match the demands of a given service – until something better or cheaper comes along.
- Another way of putting this is to say that a component involves a **relationship** between a service (in the Supply ecosystem) and a software device (in the Device ecosystem). The device implements the service, the service specifies the device.
- For many purposes, however, it is useful to fall in with conventional idiom: to talk about components as if they were objects.
- Components follow a different logic in each of four ecosystems: Device Supply, Device Use, Service Supply and Service Use.
- The dominant players in the component marketplace will be those that can understand and engage with multiple perspectives, and can straddle multiple ecosystems.

Motivation – Conflicting Notions and Perspectives about Components

Component-Based Development is commonly described in terms of a set of notions (interface, service, encapsulation, reuse, plug-n-play). But these notions do not have a single interpretation from all perspectives.

Take **reuse**, for example. Some people think reuse is terribly important, and other people don't care a fig for reuse, but do care about critical mass or quality. Champions of reuse try to engage wider support for reuse initiatives by arguing that reuse effectively means higher software quality and consistency, lower software costs, faster delivery and/or greater connectivity. But when you link reuse with these other notions, you alter the notion of reuse itself. This fact only becomes evident when you try to agree how to measure and manage reuse. A software engineer whose main motivation for reuse is to increase the productivity of software development doesn't want to measure and manage reuse in the same way as a software engineer whose primary concern is software quality or maintainability.

Even the notion of **component** itself means something different, according to whether you are talking to Java programmers or respository managers or potential purchasers. How many components are there? What are we counting: interfaces, services, lumps of code, packages, installed instances, or something else?

Copyright © 2000 Richard Veryard. All rights reserved.

Many experts will confidently tell you: components are this and not that. When faced with differences in terminology or thinking, many engineers immediately assume that the only solution is to agree a standard terminology. In other words, the software industry must have only one notion of component or reuse. Although this seems reasonable in theory, practical experience indicates that the process of consensus-building and standardization is usually fraught with conflict, delay, compromise and confusion.

We take a different approach. There are many stakeholders with different perspectives on components. We can observe that there are several competing notions of component, reuse and other key terms, and we may assume that all of these notions are valid in some context. Our goal is to understand these notions, and find ways of building useful bridges between them, not to decide which of them is the "best" or "most valid".

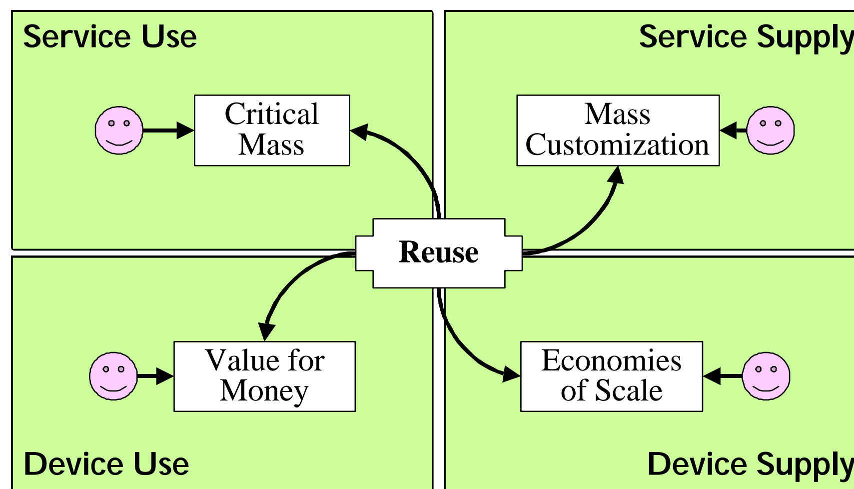

Figure 1 Four perspectives on reuse

We analyse these differences by defining multiple **ecosystems**, each following a different logic. In this book, we define four ecosystems: Service Use, Service Supply, Device Use and Device Supply.

Now we can start to be more precise about different perspectives on, say, reuse. In each ecosystem, there are different reasons why reuse might be directly or indirectly important to stakeholders in that ecosystem. See Figure 1.

What is the connection (if there is one) between reuse in the Device-Supply ecosystem and critical mass in the Service-Use ecosystem? There are lots of ungrounded claims that more reuse leads to greater quality, but where's the proof, and what would actually count as a valid proof?

In short, how can a software engineer persuade a business manager to invest in "reuse", when they don't share a notion of what reuse is, and what value it might have?

Reuse of software assets primarily makes sense within the Device Supply ecosystem, although it also has relevance within the Device Use ecosystem.

Within Device Supply, reuse equates to economies of scale in software development and maintenance. Within Device Use, reuse equates to economies of scale in software procurement and operation, which is not the same thing. These impact the

Copyright © 2000 Richard Veryard. All rights reserved.

Service ecosystems only indirectly, to the extent that they affect service variety, cost and quality of service.

Within the Service ecosystems, a different notion of reuse can be focused on the commonality of services and interfaces. In order to exchange word processing documents with my friends and associates, I need a common exchange format. It ought not to matter to me what version of what word processing product they are using, as long as the formats match. I can certainly send faxes to people without knowing what fax machine they have.

Joined-up thinking about components and related questions.

Should the software engineer adopt a business management notion of reuse, or should the business manager understand the technical notion of reuse? Neither of these – instead, we need to find ways of connecting these two notions of reuse together.

When faced with conflicting terminology, most people try to smooth out the conflicts and agree a single homogeneous set of terms. This approach has at least four potential dangers.

1. The agreed terminology becomes so bland, and so abstract, that it becomes practically meaningless.
2. The agreed terminology becomes so complicated, that it becomes practically unusable.
3. The agreed terminology leaves out some perspectives or stakeholders.
4. The process of agreement is too slow, and is overtaken by events. (For example, unilateral action by a major vendor, or the arrival of the next technological wave.)

Rather than smooth out the differences, our approach is to understand them explicitly, and to build bridges and connections between them. This is not just an intellectual exercise but an important commercial one.

The dominant players in the component marketplace will be those that can understand and engage with multiple perspectives, and can straddle multiple ecosystems.

Hybrids and multiple contexts

There was a guy who achieved some notoriety in the patent profession – my father was a patent agent – by taking out patents in strange hybrids. For example, he got a patent in a device that was a combined nuclear fall-out detector and catflap. (Given that patent law is designed to prevent silly patents being granted, this required an excellent knowledge of patent law, as well as extraordinary skill at drafting.)

Component-based development (CBD) is a similar hybrid, in the sense that it yokes together disparate concepts and mixes metaphors. Furthermore, many of the so-called gurus seem unaware of this. Endless arguments about what exactly a component is, or how you measure reuse, cannot be resolved without recognizing that there are multiple contexts.

To help make these contexts explicit, I have developed the model of four ecosystems described in this chapter. This should provide a decent basis for saying what CBD actually is – or even defining some other, more coherent notions. It also helps us to

Copyright © 2000 Richard Veryard. All rights reserved.

build conceptual bridges between the different perspectives, and find practical ways of collaborating across multiple ecosystems.

What is a Component?

Let's start with components in software, because this has been subject to a much recent discussion. There are several different sectors of the software industry, each operating with a different notion of what is a component.

Some people have an inside-out definition – a component is essentially a lump of software with certain properties. And some people have an outside-in definition – a component is essentially a set of services accessed through a specified interface whose implementation satisfies certain properties.

Business components also suffer from the same ambiguity, although this is not formally developed to the same degree as in the software industry. Some people will use an inside-out definition, resting on some notion of capability, while others will use an outside-in definition, resting on some notion of service.

Obviously if you show the same configuration to these people and ask how many components can you see, how much usage or reuse has been achieved, you will get quite different answers.

These different notions of component can only be reconciled, not at an abstract theoretical level, but through practical engagement with the business and technological drivers of a specific project or situation. The focus is not on the identity of the component, but on the act of componenting: the (always provisional) declaration that a given lump of business capability, or a given lump of software, suitably wrapped, shall match the demands of a given service – until something better or cheaper comes along.

Another way of putting this is to say that a component involves a **relationship** between a service (in the Supply ecosystem) and a software device (in the Device ecosystem). The device implements the service, the service specifies the device.

This is a many-to-many relationship. One service may be implemented several different ways, by different devices. One device may satisfy many different specifications, describing different services, accessed via one or many interfaces.

In practice, components often fall short of this ideal definition. It may be more accurate to say that the device claims to implement the service, while the service tries to specify the device.

Encapsulation

Software engineers frequently talk about a property called **encapsulation**, which implies that there is an opaque and impenetrable skin around a lump of software.

But this is a misleading way of talking about encapsulation. Encapsulation is not an attribute of a lump of 'code' that makes it into a component. It is a characteristic of a relationship between a lump of 'code' and a person or role. This or that control is accessible to the end-user, these workings are not directly accessible to the end-user.

Copyright © 2000 Richard Veryard. All rights reserved.

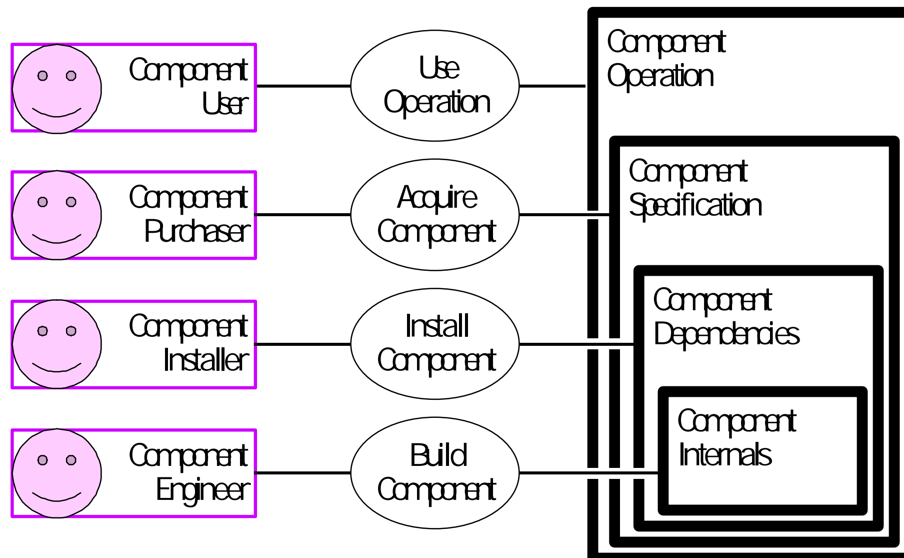

Figure 2: Four Levels of Encapsulation

Figure 2 shows four levels of encapsulation.

- The unit of use is the operation. The user of a component (which may be human or software) only needs to know the specification of the operation(s) being used. The user doesn't even need to know that several operations are actually performed by a single component.
- The unit of delivery is the component. The purchaser of a component needs to identify and acquire whole components.
- If a component calls other components, then the availability of these other components is a precondition for a successful installation of the component. The installer, therefore, needs to know what calls are made to other components.
- Most of the people who interact with a component can be satisfied with the above. The only person that needs to look inside it is the software engineer, who is charged with building, inspecting or modifying the internals of a component.

Let's Pretend that Components are Objects

It would be pleasant to imagine that somewhere in the world – perhaps in the East, wherever that is – people are more focused on relationships than things. In the West, we seem to be obsessed by things.

Materialism is not just a matter of wanting to possess things, although that's certainly part of it. It's a matter of perceiving the world as if it were composed of things. Children are taught this from an early age: most of the available books for toddlers have one word on each page, and the word is a noun: ball, bear, banana.

> Do you think that anyone has made a conscious decision that toddlers should start their acquisition of language by learning the names of things, or is it just something that happens by default? What is the alternative?

If you really make an effort, you can find books showing activities (bathing, building, blushing) or spatial relationships (inside, outside, upside down) or even feelings (happy, sad, tired). But it's still difficult for us adults to escape from the materialist mindset, or to avoid transmitting it to the next generation. After all, materialism is embedded in the structure of the book (one page, one picture, one word), together

Copyright © 2000 Richard Veryard. All rights reserved.

with the implicit notion that the child's task is to accumulate vocabulary, one word at a time.

That's why it's so difficult to see the world other than as objects, and why the object-oriented paradigm is so attractive, especially when there are excellent techniques for creating objects out of processes, out of relationships, or perhaps even out of nothing. Philosophers and software engineers have a word for this; they call it **reification**.

When relationships are regarded as things, this usually focuses attention either on the bridging mechanism, or on a static snapshot of the relationship, as for example represented by a legal contract. When processes or services are regarded as things, this usually focuses attention on the deliverable or end-result, as shown in Table 1.

Planning as Process Making scheduling and resourcing decisions in response to changing events	→	**Plan as Record** A consistent set of schedule items and resource assignments.
Negotiation as Process Ongoing negotiation and development of the terms of business.	→	**Contract as Record** A legally binding description of the relationship between two companies at a particular time.
Information as Process Selection, interpretation and dissemination of relevant business data.	→	**Information as Document** Formal results of the selection and interpretation of data.

Table 1 Regarding processes as things

The object-oriented way of describing components is extremely useful, especially for designing and managing components. It is also useful for describing the behaviour of components, and their performance in complex environments. But there are limitations to an object-oriented view of systems and components.

Further Reading

Albert Borgmann, ***Technology and the Character of Contemporary Life.*** Chicago University Press, 1984.

Francisco Varela, ***Principles of Biological Autonomy.*** New York: Elsevier Science Publishers, 1979.

Richard Veryard, ***Plug and Play: Towards the Component-Based Business.*** Springer-Verlag, 2000 (forthcoming).

Richard Veryard, "Reasoning about Systems and their Properties". To be published in Peter Henderson (ed), ***Systems Engineering for Business Process Change Volume 2***, Springer-Verlag, 2001.

Copyright © 2000 Richard Veryard. All rights reserved.