How useful are models in EDOC?

Alan Cameron Wills
TriReme International Ltd
alan@trireme.com

Alan is a principal consultant with TriReme, specialising in modelling and design methods. His book (with Desmond D'Souza) *Objects, Components and Frameworks with UML: the Catalysis approach* [Addison-Wesley 1998] has been well received as the first method specifically tackling some of the issues of design for component based development and enterprise integration.

This is a position paper given at a panel session at EDOC'99 (Mannheim).

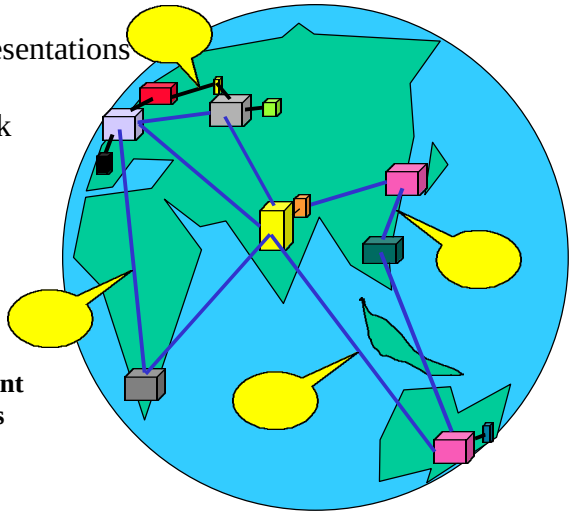
(EDOC = Enterprise Distributed Object Computing)

More information is available at http://www.trireme.com/

Connecting Disparate Systems

- Mergers, expansion, and parallel development lead to diverse systems that are subsequently interconnected
- Different internal representations
- Need to make them talk effectively
- Common open model about the business required
  - **defined in a language independent of the different implementation languages**

Cost of not doing so?



One of the perennial problems facing large organisations is keeping their various systems and departments coherent. Typically, a very large corporation is always in the throes of taking over or merging with some other company: the different cultures, technical dialects and business procedures are the biggest obstacle to integration. The result of a merger should be to economise on those features that are common to the mergands[?]; but this can't be done until they have an agreed notion of what a Customer is, what the procedures for handling an Order are, … or whatever their business is about.

The problem is most evident and tangible in the computer systems, which can't be made to interoperate until they have been adapted to a common model. The problem is primarily about making the business departments communicate --- their software just supports the business. On the other hand, people are more adaptable and it can certainly take longer to adapt the software than the staff.

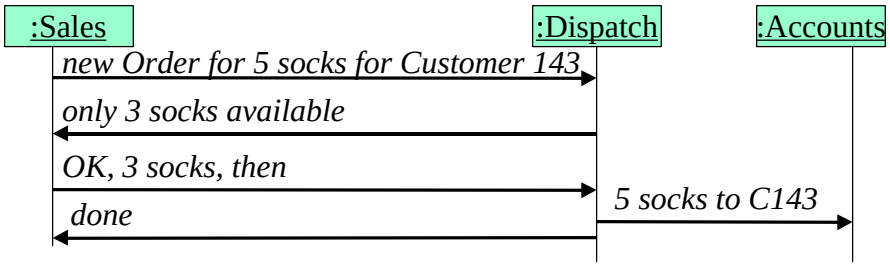
There is no avoiding the fact that hard work has to be done to make systems and business divisions talk to each other effectively: deciding protocols, the basic communication mechanisms, the formats of information transferred.

It's even more daunting to realise that this is not a one-off job: the corporation is always changing. Any viable solution must not only tackle the situation now, but must also make it easy to keep on introducing new components --- and with minimal dependence on any central authority.

Part of the solution is to introduce a single extensible language of intercommunication between the components (software or liveware). Rather than inventing a new point-to-point protocol for each possible pair of systems that need to talk to each other, it's a lot less work to have a global language.

Common Open Language: What do they say to each other?

- **Protocols of transactions**
  - Collaborations/communities
    - sequences, activity+swimlanes, statecharts, **regexps**...



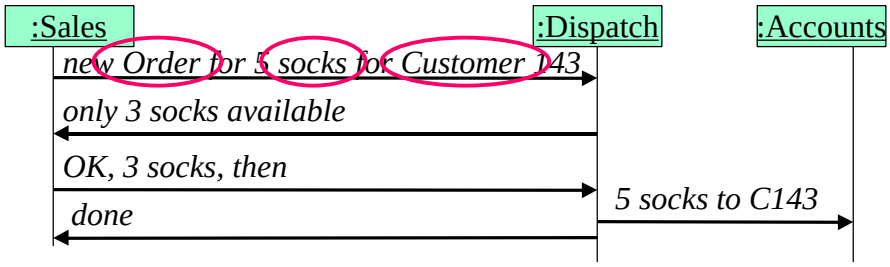
Let's think about some of the decisions the corporate architects have to make, in order to design this common language.

The technological basis is one issue: how are the systems connected together --- TCP/IP or OSI? CORBA or COM? Plain old FTP? A guy on a motorbike with a card deck? And what will be the format of the data --- flat files? Serialised Java beans? XML? Choosing among these options is becoming less difficult, especially since the competing technologies tend to converge on a common set of features. The answers tend to be the same across different business domains, too.

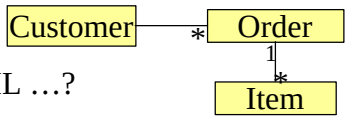
More specific to the business, is what the components say to each other, and the sequences in which they say things. Do we have a facility for an Order to be raised in one component and fulfilled in another? If so, what is the protocol for communicating that? Is the Order always accepted, or does the order-taker check with the order-fulfiller first? Etc. This depends heavily on the way in which the business is run --- which comes before the software.

The architects' job here is to define the allowed sequences of subtransactions: the grammar. Using UML, sequence diagrams are useful for simple protocols; statecharts or activity diagrams are more useful for complex ones.

Common Open Language: What are they talking about?



n **Stuff they talk about**
- Parameters of transactions
- Classes, attributes and associations; XML …?



- Business contstraints

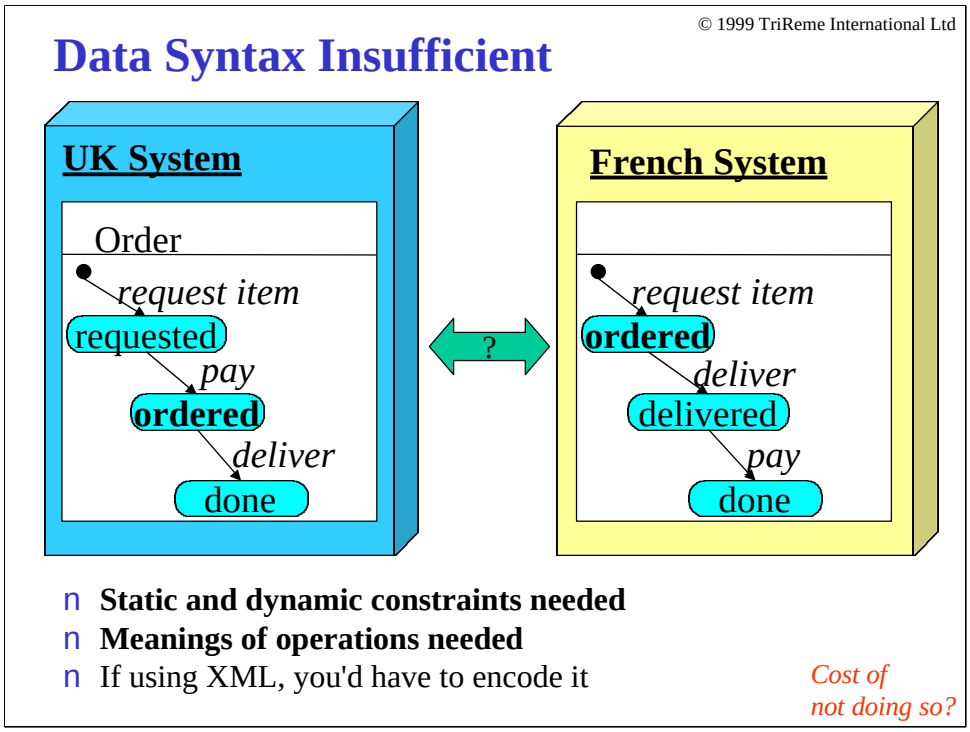
Then we look at the parameters of the messages/transactions: what are they talking about? What is a Customer, an Order…? Is an Order always for one Customer or several? Is a payment always attached to a single order, or can it span many? And so on. These relationships can be documented largely with UML class diagrams, or can be defined in XML.

The more complex business rules need additional statements about the allowed combinations of associations. These can be written in plain language or a formal dialect such as OCL (the Object Constraint Language addendum to UML). Whatever the syntax, the statements use the vocabulary defined by the associations and attributes, to say what is and isn't allowed.

By this means, we define the concepts and rules that should be adhered to in any communications between the components in our enterprise.

Remember that we're defining a protocol that has to be followed by many different systems that may themselves be written in different languages and have very different histories. Our purpose cannot be to define software: it is to define common models of the information that they exchange.

Each component will, for historical or sound local reasons, have its own internal model of the business concepts and processes. Most components will only deal with one aspect, such as accounting or dispatching or sales. So we have to be able to map the corporate model onto the individual internal models --- a job in practice done by the 'wrapping' or 'adaptor' façades that form each component's communications ports to the world around it.



Actually, it isn't enough to document just the attributes of the information being transmitted. We must also say something about the operations that can be done to them.

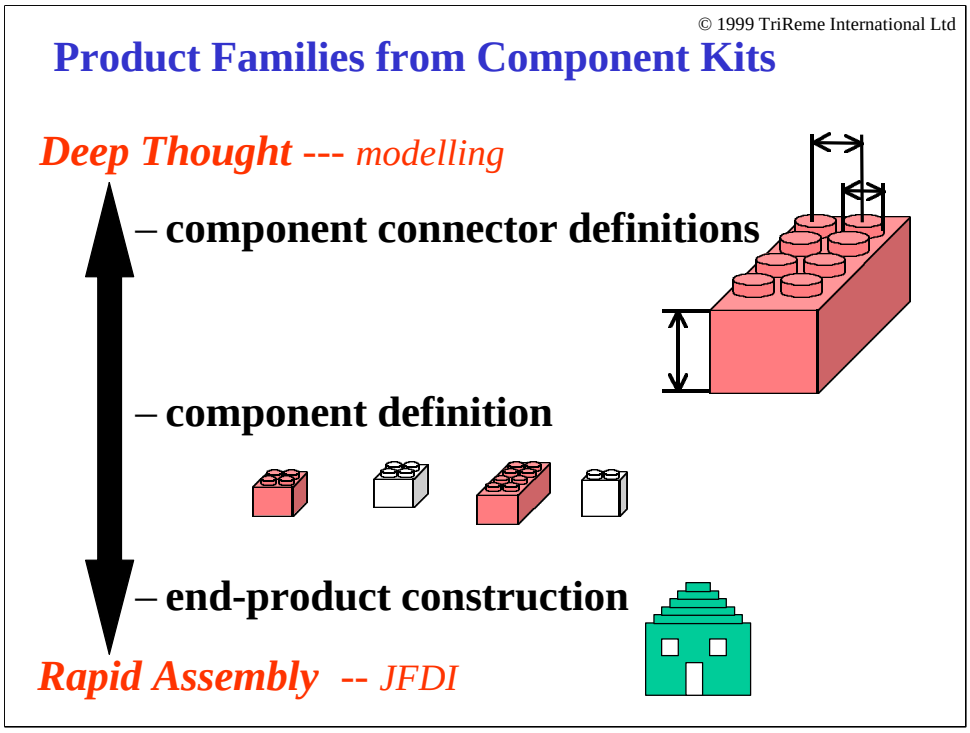
I had a client who had subsidiary companies in the UK and France. They had evolved different business procedures over the years. One day they decided to permit orders to be taken in one country and fulfilled in the other.

Looking at the two countries' *data* models of Orders, Customers, etc, they all had the same fields: name, item list, account paid, etc. The slight differences in internal representation could be resolved by some translation software in the communication path. So they thought the scheme would be easy.

But in the UK, the customer pays before the order is considered a proper order; in France, an order is an order as soon as the customer makes the request, and payment happens after satisfactory delivery. So when the first French order was transferred to UK, the French assumed it would be paid for later and the Brits assumed it was already paid --- very nice for the Customer! In the other direction, Customers were asked for payment twice.

Looking at the statecharts, the problem is obvious: they have different models, so something has to be done to integrate them. Looking at just the data fields, the problem isn't there. It isn't even visible in any invariants. It's only when you consider the operations and their effects, that the problem is visible.

Moral: a corporate data model must consider the effects of operations on each data type. In other words, it must be an object model, not just a data one. Again, we're not talking about implementations here, because they'll all be different and perhaps very historical, and perhaps partly manual.



There's another, though related, arena in which we need models that are about interoperability, and separate from any one implementation. This is in component based development: the skill of making variable families of software products from kits of reconfigurable components.

CBD architecture is like Lego or logic chips or modern car design. We plan a kit of components which can plug together in a variety of ways to make many systems. Or less ambitiously, we design one basic product, but provide plug-in components that can be substituted to change particular behavioural features. Many well-known products follow this pattern, such as Netscape, Photoshop.

CBD implies a separation of programmingroles. In end-product building, your aim is to understand the needs of the users and satisfy them by wiring components together very rapidly. There are languages or tools to help, such as the visual builder tools for Java beans and various user interfaces; or Unix Shell language.

Component design is more thoughtful. The aim is to satisfy a longer range demand for a component that can be used reliably in many different configurations. You don't know what other components yours will be plugged into, so you have to be careful to adhere to interface specifications. (Different from designing in a traditional modular environment, where you know who your neighbours are.)

Even more careful is kit architecture: the definition of the transactions that can go on between the components. This is not just a matter of defining object messages: all the things we've said about defining protocols and conceptual models and rules and operations, apply here too.

Abstract definitions are important for CBD

- Protocols hard to visualise without charts
- Permitted/mandatory sequences hard to visualise without statecharts, activity diagrams (or something similar)
- Static relationships and constraints hard to visualise without diagrams

- **Need to integrate with testing strategy**
  - Connector definitions form basis of testing
  - Components packaged with test apparatus

To integrate large enterprises --- the businesses themselves, not just the software components --- we need to understand how to make models: not just data models, but object-oriented models, containing stuff about the operations and the business rules.

For component based development to produce really flexible designs, we need to be able to do the same kind of modeling; at heart, it's the same problem.

We also need the skills and technology to be able to check whether a new component conforms to the defined model.

These are some of the issues tackled by the Catalysis approach (of which I am joint author).