Asset Mapping – developing inter-enterprise solutions from legacy components

Peter Henderson
University of Southampton
peter@ecs.soton.ac.uk
February 2001

Introduction

Contemporary, business-to-business inter-enterprise IT systems are usually based on some form of asynchronous messaging between applications. These applications talk to nearby databases in a transactional way. Designing such systems is the topic of this short paper. We pay particular attention to the fact that what businesses need is safe, evolutionary change and what businesses have is a large investment in legacy applications.

We suggest a method of describing inter-enterprise systems that lends itself well to the task of designing evolutionary developments at both the business analysis level and at the highest technical architectural level. It is here that many of the key strategic decisions about investment are made. Our purpose is to provide the means whereby those decisions might be better informed.

We make some significant assumptions. The kind of components that we imagine such systems being engineered from are large. Typically they will be applications, or databases, or more likely an encapsulated business activity comprising a number of applications coordinated around one or more databases. The example we shall use in later sections of this paper is that of a retail business activity. A typical component in that example is a (virtual) shop that we describe as four applications around one database (take a glance at Figure 2, which we will describe fully later).

Because the basic component that we propose to build from has the characteristics of being active (it comprises one or more applications), being persistent (it comprises one or more databases) and being large, we shall use the term *capability-server* for it, to draw attention to these significant aspects. If we just used the term component or application, we might convey the impression that that we recommend the technique for use at a more detailed level than we shall address here. The term capability-server is intended to capture the notion that *capabilities* are being provided, where a capability is access to information or to processing.

Another significant assumption we shall make is that capability-servers communicate with each other by sending (probably very rich) messages *asynchronously*. That is to say, the messages may take some time to arrive. The sending capability-server will not wait for a reply but will be able to service a reply, if any, asynchronously at some later time.

We conjecture that capability-servers, communicating asynchronously in this way can be used to map either the business processes to be described, or the highest level of system architecture that we (as system architects) believe will implement the business processes. These models may or may not be the same.

We introduce a type of model, which we may refer to either as a *services diagram* or a *messaging diagram*. A service is something provided by a capability-server, which is usually manifested as a set of messages. For example, the retail system will provide a purchasing service, manifested as messages for ordering, payment and delivery. A service appearing on a diagram is more abstract, a message more detailed.

In practice, a realistic system will be described by a set of diagrams, where both services and messages may appear on the same diagram. However, it may be convenient to summarise the overall structure of a system in a diagram in which only services appear, and to use diagrams in which only messages appear when detailed behaviour is being described. The rules that each diagram element must obey will be addressed separately.

The notation described in the remainder of this short note is intended for use by Business Analysts or Systems Architects at the early stages of designing an enterprise scale IT solution. As we shall see, the notation can be used to produce a map (diagram) of either Applications or Business Components, showing in particular how they communicate with each other.

More importantly, it can be used to capture knowledge about the behaviour of legacy business processes and/or legacy applications where the intention is to re-engineer or re-deploy these assets in order to determine opportunities for business evolution and innovation. When used in this way, we refer to the modelling process as Asset Mapping or Legacy Mapping.

The method can also be used to specify requirements for new business processes and/or applications. In the next section we give an example and describe the notation. Later sections will describe how it fits with other notations, in particular UML, and give some architectural guidelines for its use.

The methods described here are nothing more than an accumulation of best practice. The sources of this practice include [1], [2], [4], [7], [9], [15], [17], [18], [21] and [24]. We have built on all these sources. We have however identified the concept of capability-server, and its participation in asynchronous message-passing architectures, as the key to modelling enterprise and inter-enterprise systems.

Asset Mapping

Figure 1 shows a (grossly simplified) retail solution. Each of the boxes in this messaging diagram is a capability-server (a business component), which can most easily be thought of as an application, or a set of applications arranged as servers around a database. So capability-server components usually have significant state.

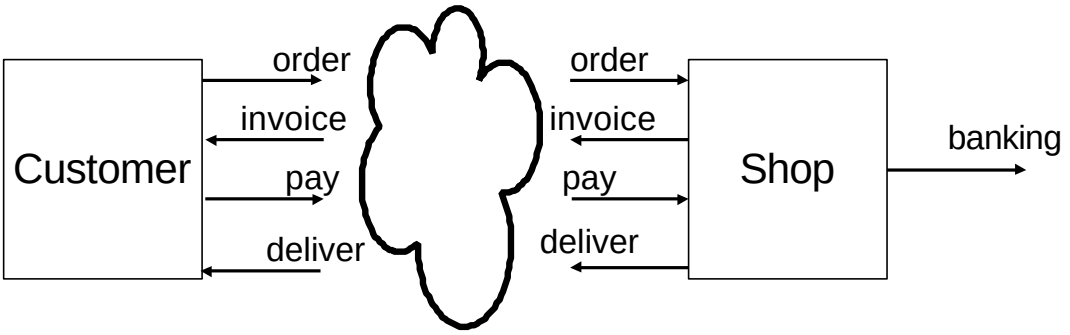

Figure 1 : Two capability-servers: Legacy Business Components in a Shopping Scenario

The arrows in this diagram are intended to denote messages. Communication between business components is asynchronous, using rich messaging (example, XML over MQ or HTTP) [6], [23]. The clouds are the communications medium, which is considered to be something that holds on to the messages while they are in transit. That is what we mean when we say that communication is asynchronous. By and large we will assume that message delivery is guaranteed (once only) but receipt is not necessarily in order of sending.

The key IT component in the example is the Shop. It is able to receive orders and payments over the *order* and *pay* message channels, (or, we could say, via the order and pay service interfaces). Its database keeps historical and contemporary information about orders and the organisation's progress in fulfilling them.

When this system is described in this way, we have only been specific about its structure, not its behaviour. The diagram might be supplemented by a behavioural description of the following sort. The Customer may place an *order*. Upon receipt of the *order* the Shop will issue an *invoice* and await payment. The Customer will *pay*. Upon receipt of payment the Shop will check validity of the payment with *banking* and, if that works out, arrange delivery.

If this is what the Customer expects, and this is all the Shop can do, then successful completion of an order is of course possible. But in all likelihood, the Customer will expect a wider range of behaviours. The Customer may wish to pay with order and not be expecting an invoice. The Customer may expect delivery to be arranged before payment is confirmed, or even before payment is made. A Shop with this added flexibility cannot mandate the order in which messages might arrive. The most versatile Shop will be able to process messages (in a business-sensible way, i.e. not just sit on them) in whatever order they arrive.

Indeed, a theme of this paper, which motivates some of the guidelines that we give later, is that capability-servers should be able to process messages in any order. Some orders might be quite unusual, such as payment arriving before an order, or delivery being arranged before an order, but they do arise in practice. They might be the consequence of innovative business practices, or they might be the consequence of system failures (e.g. unusually long message delays). They might lead to inconsistency in the information held by different capability-servers in the system. If so, the system as a whole must be tolerant of this information inconsistency. We will discuss later the extent to which capability-servers can be made *inconsistency tolerant*.

It is probably worth considering why we have used the term capability-server for the building blocks in our solution. In what sense is Shop a capability server? More to the point, in what sense is Customer a capability server? The Shop, as we shall see, is an encapsulated Legacy system that heretofore offered a capability, which was to track orders, payments and deliveries entered into it manually. By encapsulating it behind a messaging interface, it now provides the extended capability of being able to operate in an inter-enterprise electronic data exchange environment. Its capability is enshrined in its persistence (it remembers the data) and in its activity (it processes requests and it progresses work).

The Customer considered as a capability-server may seem a little more unusual, since we think of them as being a person at a keyboard. But they may as well be an application acting on the customer's behalf: an agent. They may as well be a set of such agents operating cooperatively on the customer's behalf. As such they will have a capability of remembering the states of various orders and of being proactive in progressing them. The Shop can view the Customer as having the capability of generating orders, processing invoices, making payments and accepting deliveries.

Along with the capability that a capability-server provides, there is a concomitant responsibility. Usually, in systems design we shall wish to place the responsibility for something in exactly one place. This is a concept to which we will return.

Let us look more closely at the Shop capability-server. This component has been mapped from a legacy IT system that comprised a back office database supporting four applications (order processing, financial processing, shipping and customer care). This Legacy System is shown in Figure 2. We imagine that the retail organisation that owns this legacy system regards it as reasonable in future evolution to consider this collection of applications as a single, reusable business component. As a single component its basic behaviour is as follows.

All information about customers, orders, stock etc is held in the BackOffice database. Upon receipt of an order, originally on paper, the *order processing* application is used to enter details of the order on the database. Somehow, *Finance* is informed of the order and they issue an invoice, updating the database accordingly. Upon receipt of payment, the *Finance* application is again used, first to check the validity of the payment with the bank, then to update the database, and possibly to advise Shipping that delivery is due. The *Shipping* application is used to interrogate the database in order to arrange delivery, which ultimately is also recorded. There is a fourth application, used by the Customer Care department, which answers mail and telephone calls concerning progress of customers' orders.

None of this behaviour of the Legacy system need change. Nor do we require any specific behaviour from its internal components. For example, it may be (as hinted at on the diagram) that *order processing* somehow notifies both *Finance* and *Shipping* that an order needs attention, or it may be that these two applications simply monitor the database as a means of progressing an order. What does matter, is that the behaviour expected at the messaging interface, which is this Legacy System's window to the e-business domain, is supported by the Legacy System (considered now as an capability-server). Thus, if the capability-server wants to allow delivery before

payment is processed, the Legacy system may need to be enhanced (loosened) to allow this.

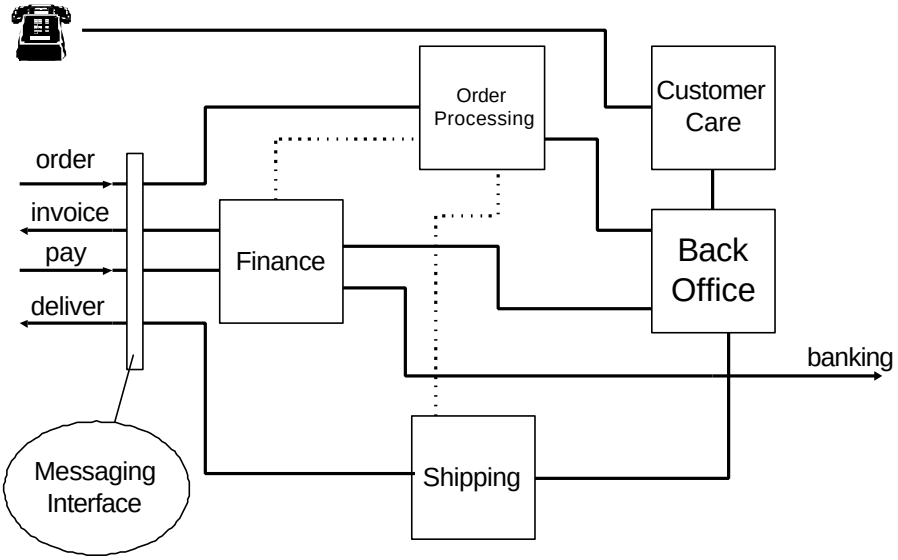

Figure 2 The Legacy System which was mapped to produce Shop

The key concept here, and the key benefit of Asset Mapping, is that the *responsibility* for providing the service is placed clearly upon one component (i.e. one capability-server). We can state clearly, as we shall see as this example develops, that each capability-server can be made responsible for

- The information which it holds
- The business processes (services) that it offers.

The Shop, for example, is responsible for

- Information: customers, orders, stock (not shown)
- Services: purchasing, customer-care, stock-maintenance (not shown)

Inter-Enterprise Systems

Let us continue to build an inter-enterprise system, based on the notion of capability-servers. We will show how (virtual) shops can be described and configured into e-business, offering added-value both upstream and downstream.

A more complex Shop is shown in Figure 3. Here we have added messaging which will allow a customer to return goods and for reimbursement to be made for that. This is a trivial extension. For that reason, we do not consider what changes might be required “in the box”. Rather, we discuss how this new Shop and this new Customer might interact with each other, with others of the same sort and with others whose interfaces are not quite the same.

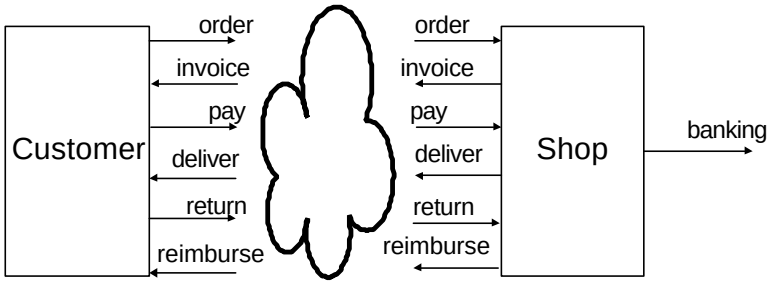

Figure 3: A Shop capability-server with an extended interface and a corresponding Customer

Again, in Figure 3, we are showing messages. It is clear that the messages sent by one capability-server in Figure 3 are able to be processed by the another. But what if we mixed the capability servers from Figure 1 with those from Figure 3. Clearly this must be something which we support, for in a flexible business-to-business environment it must be possible to launch any new service and have it interact with existing ones to the fullest extent of their shared capabilities (but see [13]).

So, what would we expect if the Customer from Figure 1 encountered the Shop from Figure 3? The Customer would not be able to make use of the *return* service offered by the Shop, and would not be able to process a *reimbursment*. But the Figure 1 Customer should be able to carry out all of the interactions with the Shop from Figure 3 that it was able to perform with the Shop of Figure 1.

Conversely, the Customer from Figure 3 would be able to make use of the Shop from Figure 1. The Customer would simply find that the Shop was not able to accept *return* messages.

This flexibility is a considerable requirement upon capability-servers. They need to be able to operate in a changing environment and not be pedantic about the availability of services. I have discussed this type of flexibility elsewhere [13], [14]. The technologies required to provide this sort of flexibility are being developed, in particular the XML-based Web Services Description Language (WSDL)[5] and the Universal Description and Discovery and Integration (UDDI) [23] specification which, between them, provide the means to describe a service and to locate providers of services.

The model we have developed so far is not inter-enteprise, unless we consider Customer to be an enterprise. But we have all the machinery we need to move to the inter-enterprise level.

Consider Figure 4. Here we have introduced a virtual shop, which we denote by the capability-server referred to as Mall. The interfaces to the left of the capability-server provide the services identical to the Shop of Figure 3. That is, the Mall supports the more elaborate Shop interface.

The Mall is supposed to be providing a one-stop service that in some way integrates the capabilities of a number of Shops, two of which are shown in Figure 4. The Customer is not shown in Figure 4, for brevity.

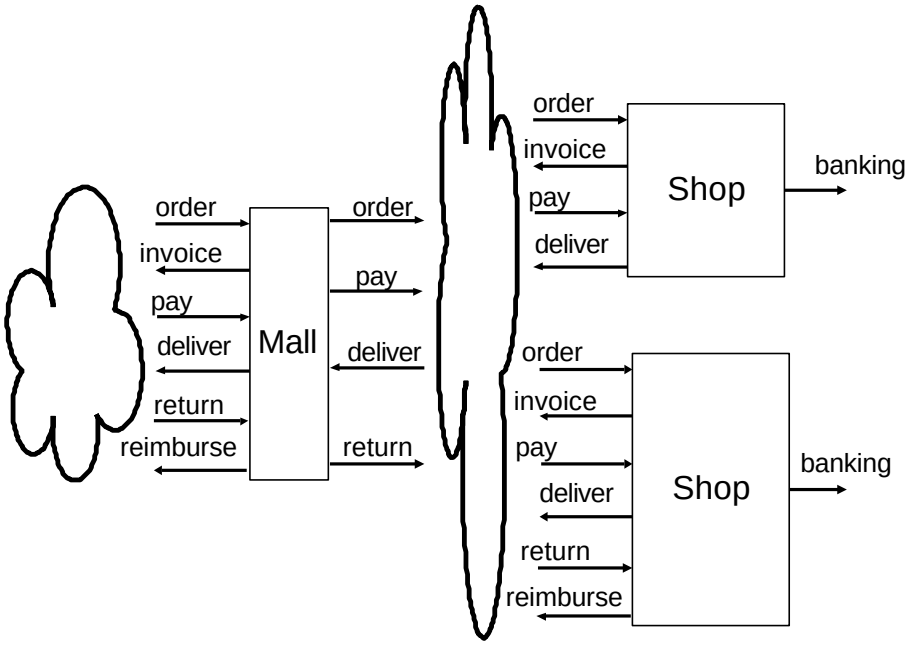

Figure 4 an inter-enterprise solution, a one-stop shop

The Mall provides added-value services to the Customer. As well as providing an integration of access to a number of shops, we can imagine that the Mall negotiates best prices with the Shops that it integrates. Equally significantly, the Mall provides a *return* service to the Customer, whether or not the Shop from which the item is actually bought provides such a service. We can imagine a number of ways in which the Mall might implement this business function.

The Mall also provides added-value services to the Shops. It guarantees payment. Indeed, it looks after all financial risk. Consequently, upon receiving an order, a Shop can immediately arrange delivery, knowing that payment will be forthcoming. The Mall pays the Shop in bulk, having kept a record of payments due and after subtracting reimbursements.

Of course, none of this business logic is apparent from the diagram (Figure 4). But the diagram does provide the means for articulating this inter-enterprise solution at the business level. It also provides the means for articulating the architectural and technical requirements. The architect of the solution will need to consider the precise functionality to be provided by each capability-server at each interface. This will best be achieved by enumerating the messages that can be expected and by describing the capability-server's response to each message.

For example, on the *pay* interface, a Shop might expect a message that identifies an order, an amount and a financial instrument for collecting that amount (e.g. an electronic funds transfer). The Shop will respond by updating its internal records to record this payment or, if there is some problem (e.g. can't find corresponding order) to take appropriate action.

This *pay* interface may be appropriate for Customers directly, but not appropriate for dealing with the Mall, where a payment will identify a number of orders. The Shop capability-server may therefore need to be upgraded. It is not the topic of this paper to

discuss how that might be done. Our purpose here is to illustrate that Asset Mapping is a technique that enables an architect to address the appropriate design issues at the appropriate level of abstraction and at the appropriate time in the design cycle.

An issue that does arise for the business analyst and/or the system architect at this stage of design of an inter-enterprise solution is whether or not a solution that they propose is valid. Will it achieve the stated objectives? Is its stated behaviour correct? It is possible to be sufficiently precise about the behaviour of systems at this level that a working model can be built and some of the validity issues addressed. We have shown how this can be done with tools we have built ourselves [10], [11], [22] but there are many tools available for simulation which can be used for this purpose.

The concepts for modelling at the enterprise and inter-enterprise level are not well developed (for notable exceptions, see [1], [17], [19] and [24]). However, the formal basis for such models is well established. In particular see [4], [16] and [20].

Being Innovative

It is not sufficient at the inter-enterprise level either just to be able to map out completely new solutions or just to map out existing legacy systems. It is essential to do both. The existing legacy systems comprise the principal assets of a business, yet the business needs to evolve and be innovative in its business domain without having to start from scratch. It is essential to realise these legacy assets as part of the new solution. The Asset Mapping method encourages the architect to describe these assets as capability-servers and to imagine them being redeployed in an inter-enterprise scenario where they communicate by sending rich messages asynchronously.

There will not be a single best solution to describing an existing legacy system as a collection of capability-servers. Indeed, the choice of which components to separate and which to keep together will be the key to providing cost-effective flexibility in the redeployment of these legacy assets.

The choice of capability-servers will also be key to being innovative in the business domain. To illustrate this, consider the (somewhat artificial) restructuring of our original shopping solution shown in Figure 5.

We have separated the delivery capability of the Shop from the order-taking capability. This is showing the Shop in an intermediate stage of evolution, where the intention is to outsource delivery to specialised distributors. The eventual business innovation involved here is for the Shop (now called Outlet) to concentrate on added-value services to customers while providing a wider range of products delivered directly from suppliers (or their Distributors).

The added-value services are as follows. Allow customers to browse our offerings. Accept orders and payments from customers. Notify customers when delivery has been scheduled. Place order for delivery with distributor. Periodically pay distributor. Accept returns from customers and make reimbursements. Batch up returned items and send to distributor(s) over their supply interfaces. Clearly, in practice, these new services would need considerable elaboration, and validation, before an effective business model was fully realised. But it does show how Asset Mapping can support

innovation in business services, while addressing the need for an evolutionary approach.

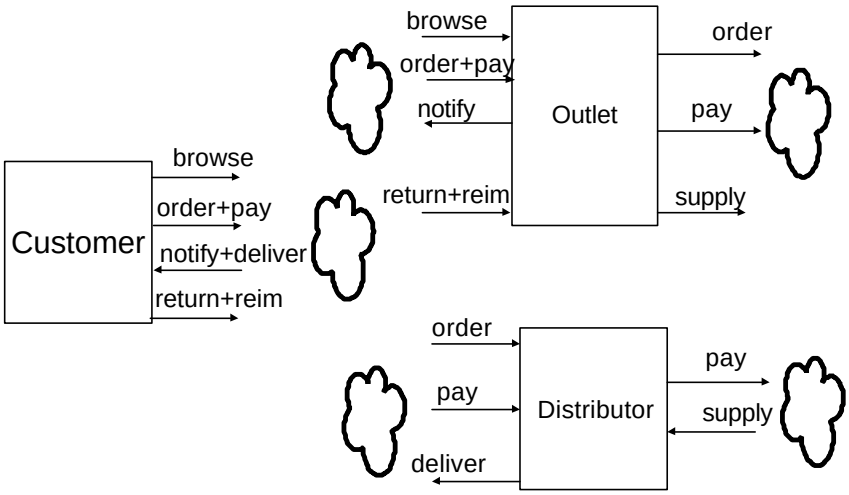

Figure 5 The Shop reorganised as two capability-servers, separating delivery capability

Notational Conventions

We are intentionally relaxed about the meaning of the arrows in the diagram. We believe this facilitates architectural discourse at the stage in a system design when major decisions are being made about which business processes to deploy and where responsibility for those business processes should be placed.

Figure 6 shows the two alternative meanings of the arrows.

Sometimes in a diagram an arrow will simply denote a channel on which messages can flow, and the label will name the kind of message sent on that channel. In that case, the box is being thought of as a business component that is collecting the messages and somehow converting them into other messages that it sends to other parts of the system.

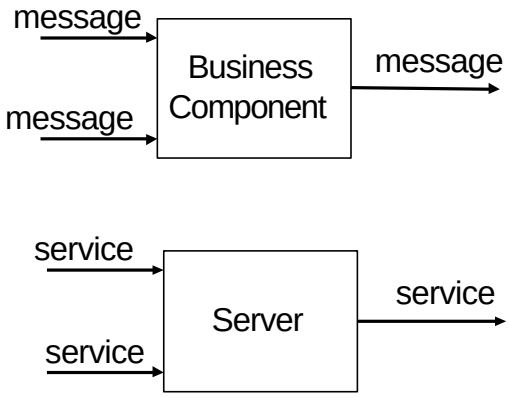

Figure 6: Capability-servers Notation

Sometimes in a diagram an arrow will denote a service (or a service interface). This may or may not be implemented by messages. Even if it is implemented by messages, it will in general be a *conversation*. That is, the arrow will effectively denote a whole set of messages flowing in both directions in order to implement the service. In this case, the recommended convention is that the arrowhead points at the server (for the service named on the arrow).

To exemplify this, refer again to Figure 5. The arrow marked order+pay could be read as a channel on which messages concerning orders and payments flow. The Outlet component converts these (eventually) to outgoing messages to fulfil the order. Alternatively, the arrow marked order+pay could be read as a service and the Outlet component considered a server for this service. By a conversation over this service interface, the server elicits information sufficient eventually to fulfil the order.

In general, during development of a model of this sort, arrows will be used in both ways, mixed on the same diagram. Talking about the behaviour of the components is easier when the detailed messages are understood. Eventually, however, it is usually necessary to simplify the diagram by bundling sets of messages together into services.

Architectural Guidelines

We give here some guidelines about what to look for when using this notation to design or re-engineer a system. The assumption is that the architects begin with some legacy from the business domain: legacy applications, assets that are to be redeployed in the new solution and legacy business processes that are the assets whereby the core business functions. These assets are to be reconfigured and new assets to be established in order to achieve the evolution being sought.

1. When a business component has been mapped, determine constraints on the sequence in which messages can be processed. Can these constraints be loosened? This may lead to more reusable behaviour.
2. Use standard UML diagramming [8] to describe the behaviour of individual components. Use Sequence Diagrams or Collaboration Diagrams as appropriate. In fact, the capability-server notation is very close to Use Case diagrams, which are well established in UML as the abstraction that is elaborated using these two types of Object Interaction Diagram.
3. Are the components that you have mapped *stateful* or *stateless*? If you are diagramming stateless components, are you at the right level of abstraction? For enterprise systems and inter-enterprise systems, mapping them as a set of communicating stateful components is to be recommended. In general, a capability-server is responsible for some information, in which case it will certainly be stateful.
4. For components with state (capability-servers), what is the logical or conceptual nature of the state that they are responsible for. Consider the nature of the updates and queries upon this state. Will it be consistent with the real world? (usually only reasonably close). Will it be consistent with other components? (usually not). Will components be moving always towards reconciling inconsistencies? (usually). For example, in the Shop, the orders on

the database will not be up-to-date with the real world. Most of the time clerks are devoting their time to editing this database.

5. For components with state, make sure the state isn't too big (all the company's data) or too small. For enterprise scale systems there will be logically distinct groupings of data (e.g. orders, finance, personnel, etc). They won't often be in separate databases. Nevertheless, try to separate these off onto different conceptual servers. It will loosen up the system and greatly enhance the prospects for re-engineering and innovation

Conclusions

We have introduced the notion of capability-server, a large business component that is persistent and active. It is an abstraction which embraces the typical IT installation in an enterprise, a set of applications configured around a set of shared databases. This IT installation supplies a business capability and communicates asynchronously via rich messages with other capability-servers. We have suggested that all Legacy IT systems can be modelled as capability-servers and have referred to this method of modelling as Asset Mapping, because it allows a business to capture a model (a diagram, a map) of their IT and business assets. This in turn enables evolution and innovation in the business domain. We have shown how this modelling method can be applied at the enterprise and inter-enterprise level and have given some guidelines for how best-practice may be applied. We believe that we have demonstrated that engineering in the business domain is not only possible but potentially highly effective.

References

[1]. Bensley E, et al, Evolvable Real-Time C3 Systems, First International Conference on Engineering of Complex Systems, IEEE Computer Society Press, 1995

[2]. Bustard D, P Kawalek and M Norris, Systems Modelling for Business Process Improvement, Artech House (2000)

[3]. Calder M and E Magill Feature Interactions in Telecommunications and Software Systems IV, IOS Press (2000)

[4]. Cardelli, Luca Abstractions for Mobile Computation *Microsoft Research Technical Report MSR-TR-98-34* available at research.microsoft.com (1998)

[5]. Christensen E et al, Web Services Description Language (WSDL) 1.0, IBM/Microsoft Joint Working Document, see http://www-4.ibm.com/software/developer/library/w-wsdl.html?dwzone=web/ (2000)

[6]. Dickman, Alan Designing Applications with MSMQ – Message Queuing for Developers *Addison Wesley,* 1998

[7]. Fowler M, Analysis Patterns-Reusable Object Models, Addison Wesley (1998)

[8]. Fowler, M, UML Distilled – Applying the standard Object Modelling language *Addison Wesley*, 1997

[9]. Henderson, P & Pratten, G.D., POSD - A Notation for Presenting Complex Systems of Processes, in *Proceedings of the First IEEE International Conference on Engineering of Complex Systems*, IEEE Computer Society Press, 1995

[10]. Henderson, Peter and Bob Walters, Behavioural Analysis of Component-Based Systems, *Information and Software Technology (2001),* available at http://www.ecs.soton.ac.uk/~ph/papers

[11]. Henderson, Peter and Bob Walters, Component Based systems as an aid to Design Validation *Proceedings14th IEEE Conference on Automated Software Engineering, ASE'99, IEEE Computer Society Press, 1999,* available at http://www.ecs.soton.ac.uk/~ph/papers

[12]. Henderson, Peter and Bob Walters, System Design Validation using Formal Models *Proceedings10th IEEE Conference on Rapid System Prototyping, RSP'99, IEEE Computer Society Press, 1999,* available at http://www.ecs.soton.ac.uk/~ph/papers

[13]. Henderson, Peter. Laws for Dynamic Systems International Conference on Software Re-Use (ICSR 98), IEEE Computer Society Press, available at http://www.ecs.soton.ac.uk/~ph/papers

[14]. Henderson, Peter Business Processes, Legacy Systems and a Fully Flexible Future, appears in [15], available at http://www.ecs.soton.ac.uk/~ph/papers, (1999)

[15]. Henderson, Peter Systems Engineering for Business Process Change-papers from the EPSRC Research Programme (vol 1), Springer UK (2000)

[16]. Hoare C.A.R, How did Software get to be so reliable without proof, *Keynote address at the 18th International Conference on Software Engineering*. IEEE Computer Society Press, 1996. see also *http://www.comlab.ox.ac.uk/oucl/users/tony.hoare/publications.html*

[17]. Leymann F and D Roller Production Workflow – Concepts and Techniques, Prentice-Hall (2000)

[18]. Leymann F and D Roller Workflow-based Applications, IBM Systems Journal, Vol 36, No 1 (1997)

[19]. Martin, M and J Dobson, Enterprise Modelling and Architectural Discourse, Centre for Software Reliability, available at http://www.csr.ncl.ac.uk/emods/ (1998)

[20]. Milner, Robin, Elements of Interaction, Turing Award Lecture *Communications of the ACM*, Vol 36, No 1, January 1993

[21]. Mulender, Sape J Distributed Systems *Addison Wesley* 1993

[22]. Phalp, Keith, Peter Henderson, Geetha Abeysinghe and Bob Walters, RolEnact - Role Based Enactable Models of Business Processes, Information And Software Technology, 40(3) (1998) 123-133

[23]. UDDI.org, Universal Description, Discovery and Integration – Technical White Paper, joint Ariba/Microsoft/IBM working paper, see www.uddi.org (2000)

[24]. Veryard, R Reasoning about Systems and Their Properties, to appear in Systems Engineering for Business Process Change (Vol 2) , meanwhile see www.veryard.com (2000)