Web Components: services that wear state on their sleeve

Steve Battle
Information Infrastructure Laboratory
HP Laboratories Bristol
HPL-2002-68
April 5th , 2002*

web-service,
e-service,
business
process,
document
state,
component

In this paper we describe a modular approach to building web services that supports collaborative, cross-enterprise modes of working. The web component model emphasises the separation of state and behaviour. Our functional components are web services, while state is reified as web resources. Where components are composed, they need not communicate directly with each other, but via shared state. The approach is motivated by a scenario drawn from the world of e-print.

Web Components: services that wear state on their sleeve

Steve Battle, March 1, 2002

Hewlett Packard Laboratories

Abstract

In this paper we describe a modular approach to building web services that supports collaborative, cross-enterprise modes of working. The web component model emphasises the separation of state and behaviour. Our functional components are web services, while state is reified as web resources. Where components are composed, they need not communicate directly with each other, but via shared state. The approach is motivated by a scenario drawn from the world of e-print.

web components

Web-services turn distributed computing on its head. Where traditional distributed systems may be characterized as closed, web services are *open*. A fundamental premise of web-services is that they may be used in ways unforeseen by their designers.

The current generation of web based services are *page centric*, in that the service provider offers only an HTML[i] user-interface to their service. This has become a hindrance to integration and automation because clients lack a conceptual model of what the service does. The *service centric* view of the web starts from a different place, by describing the functionality that is on offer. We interact with web services by exchanging XML[ii] documents that convey business information; enquiries, orders, quotes, etc. XML has the advantage of being a non-proprietary and open format. By nature, XML is extensible, meaning that developers are empowered to build their own customised XML documents to serve their immediate business needs, rather than relying on a few powerful vendors.

Web-services emphasize the separation of concerns between form and function. Once a service is deployed, we can still bolt on the familiar web based user interface. Indeed it now becomes possible to think about supporting a host of interfaces, making the service accessible from a wide range of emerging information appliances. The interface can change *without changing the underlying service*. The last phase of the web saw a sharp distinction between the "dot.coms" and traditional industries. The future may see a greater blending of "clicks and mortar" businesses able to integrate a diverse range of customer on-ramps, "call,

[i] Hyper-Text Markup Language
[ii] eXtensible Markup Language

click, or come-in"[1], letting the customer decide whatever mode is best for them. The physical storefront is as valid an interface to the web-service as the browser or phone call.

However, web-services are not just customer facing. Behind the storefront, web-services will allow businesses to collaborate on joint activities that cross enterprise boundaries, with each participant focusing on their core values[2]. Collaborative activities are often task-led, involving knowledge-rich human interaction and sustained business relationships. They may be less centralized and prescriptive, but more ad-hoc and adaptive. Unlike traditional hierarchical organizations where strict top-down lines of control are maintained, collaborative networks are more opportunistic and responsive to change. Indeed, change is seen as part of the process[3].

Web services are often described as a component model for the web. Component-oriented programming is about assembling systems from prefabricated parts, designed to be modular and re-usable, using common communication protocols, are user configurable, and easily composable. In this paper we explore a component-oriented approach that does not encapsulate state and behaviour within a single entity like object-oriented systems, but deliberately keeps them apart. In this service-oriented world, behaviour is provided by services (our functional components), while *state is objectified as a document*. Looked at another way, the function of the component model is to *add behaviour to documents*. Our approach is similar to that of the Process Wall[4], which demonstrates the advantages of the state-server approach in heterogeneous computing environments. Just as XML is the foundation of document exchange, a defining feature of web-services, so XML is the basis of *document state*[5]. A web-service description typically ends with a definition of the service interface using languages such as WSDL[i,6]. The document state model seeks interoperability not only at the level of input/output behaviour but of the business model these services operate upon. With today's n-tier architectures the storage tier remains within the sphere of control of a single enterprise, hidden behind the application server. Our approach inverts this picture, moving *shared* data out onto the web. These independent web-resources may represent client data owned and managed by the client, or may be collaborative data managed by a third-party.

As industry commentators have noted[7], the challenge of 'enterprise *information* integration' is now becoming more important than the 'enterprise application integration' solutions offered by current web-service solutions. The promise of web-services lies in creating a flexible infrastructure that merges the information models

[i] Web-Service Description Language

of enterprises collaborating over the web, using this to drive flexible, decentralized business processes.

eBusiness process

We explore the document state approach using a scenario drawn from the world of e-print; the business of delivering high quality print to the marketplace via the web. Print cannot be treated as a commodity that can be easily bought and sold over the web; it is fundamentally about the interaction between the client and the printer. The byword is complexity, as customers tend to outsource their most difficult jobs. Even then, print is more than just delivering a file to a printer. The print process is highly collaborative, involving the customer and print provider with print management companies, agencies, and print brokers. Customers will typically work with a small number of service providers who may be tightly integrated into their workflows.

Our first step is to capture the intended business process. A business process is a market-centred description of an organization's activities, designed to fulfil a business contract or deliver value to the customer[8]. In other words, it defines what a process achieves, rather than how to do it. We are not concerned here with exactly *how* these use-cases are realized. An important principle of service-oriented design is that we hide as much as possible about the implementation and technologies behind the service. The choice of programming language, platform, operating system, workflow engine, or back-end database is simply not an issue for the user of the service. Our main concern here is with the question of "What information can I share with my partners?"

The Unified Modelling Language[9] (UML) is a visual modelling notation that can be used throughout the project development lifecycle. It is a set of notations that can be employed at different stages of development, from requirements capture and analysis, through implementation to deployment. In the most part, practitioners of UML recommend a use-case driven approach to project development. Use-cases are essentially a way of capturing the high-level activities the proposed system should support. Use-cases provide the starting point for a modular representation of business process by laying out a coherent set of activities and their associated actors.

Use-cases are compatible with our service-oriented view. They define an *external* view of the system. We need only be concerned with what services are available and how they are used. Both use-cases and services are described in terms of the *value* they generate for both the customer and the service provider; value flows both ways. Figure 1 includes a number of use-cases drawn from the print scenario. While a single 'print' use-case can be seen as a high-level use-case from the point of the client, it would fail to describe the value of the service to all of the participants involved, and who contribute to different phases of the business process. Instead, we take a first-level cut at the *business* use-cases[10]. Descriptions of each use-case are provided below.

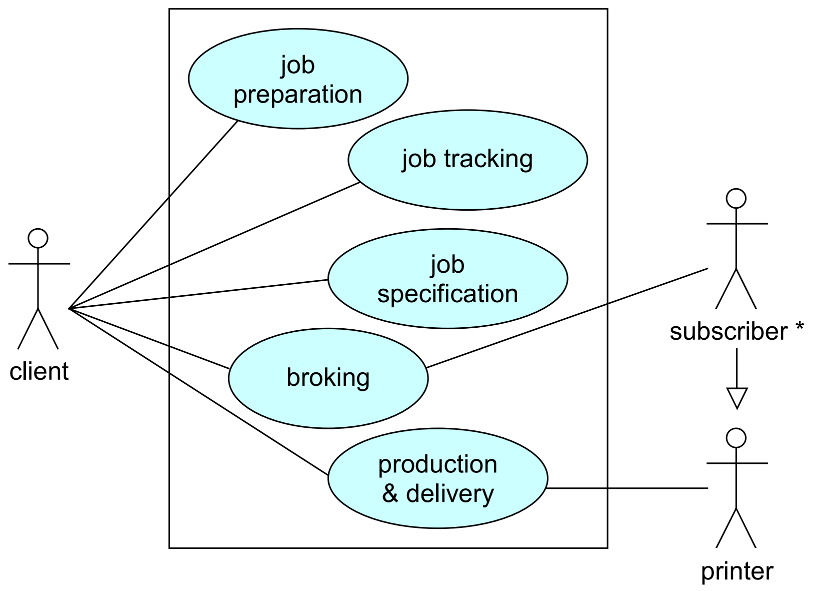


Figure 1: e-print use-cases

The use-case diagram introduces a number of actors that represent various roles within a use-case collaboration. The communication lines between actors and use-cases tell us who is involved in a particular activity. The fact that subscribers are printers is indicated by the generalisation relationship between them. An actor may be realized by many participants; there may be many subscribed printers (as indicated by the asterisk) who may be the preferred suppliers of the client. A single participant may realize multiple actors. For example, at the end of the broking stage, the client will assign the job to one of the subscribed printers.

1. **job preparation**
   The artwork is prepared and a new job is created, including a manifest of artwork resources. The client is able to view and manage the artwork, which may be archived for future re-prints or for re-purposing.

2. **job tracking**
   The client is able to track the progress of the job throughout the print process.

3. **job specification**
   Based on an analysis of the job, its main features are captured in a job-specification that includes the nature and size of the job. Additional

information may be supplied, including any special finishing required, and the timescales on which it is to be delivered. The analysis may include pre-flight testing designed to detect common printing errors before the job proceeds further.

4. **broking**
   A request for quotes (RFQ) is published and subscribed printers with matching offers are notified. Price estimates are calculated by the printer on the basis of the information within the RFQ, considering the cost of the materials required, and how the schedules fit in with their own work-load. The printer may be working in an existing relationship with the client that may influence the price.
5. **production & delivery**
   After selecting a printer, production is carried out by the printer according to the job specification. The job is then shipped according to instructions; it may be shipped back to the client, or it may require a direct mail-out.

There is an implied ordering over the use-cases described, with each one establishing the conditions for those that follow it. For example, a job quote can only be issued when that job has been properly specified and a request for quotes has been issued. UML lets us capture the pre- and post-conditions from which these dependencies can be inferred. It is this network of dependencies that defines the business process.

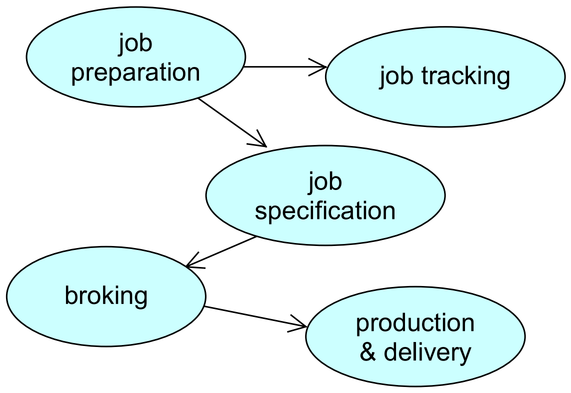


Figure 2: network of precedence relationships

Figure 2 attempts to capture this network of causal relationships. The provenance of the *precedes* relationship stems from the Open Modelling Language, OML, developed by the OPEN (Object-oriented Process, Environment and Notation[11]) consortium, and already has some proponents in the UML camp[12]. Using this additional notation we can now represent the causal precedence relationships that were implicit in our original use-case diagram. These provide an explicit indication of the ordering over use-cases without going into details of specific pre/post-conditions. Precedence relationships do not denote the flow of control as such, but the *flow of information*. Information flows being far more suited to collaborative working than traditional methods of command and control.

We understand these use cases as operating upon a set of objects that represent shared state, as shown in Figure 3. They are represented as web-based documents or document fragments; *document state*. Many of these objects are aggregated to form composite objects such as the job. These objects (their schema) form the vocabulary in which we express the pre- and post-conditions for each use-case.

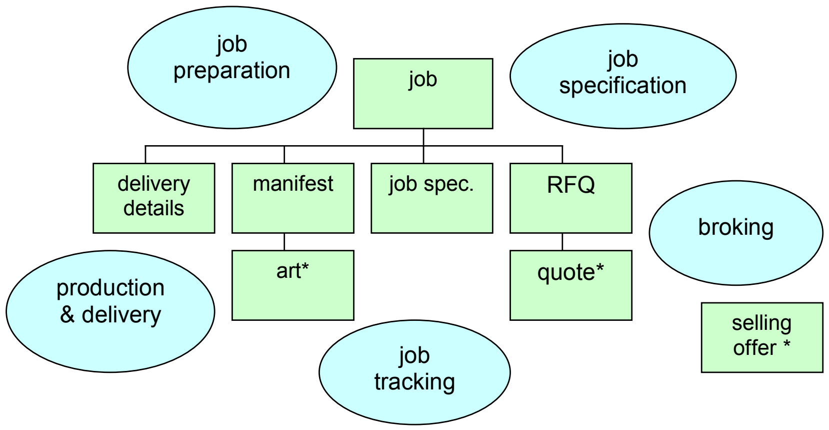


Figure 3: shared state

The intended meaning of these shared objects is as follows.

- **job**: A composite object combining content with negotiation and performance data.
- **manifest**: The set of resources that comprise the content of the job.
- **art**: reference to a specific item of artwork that forms part of the job.
- **job spec**: a detailed job specification suitable for guiding production.
- **RFQ**: Request For Quotes. A buying offer summarising the job specification. Also used here to collate returned quotes.
- **quote**: A quote is created by a subscribed printer in response to an RFQ detailing the price and timescale on which they can perform the job.
- **delivery details**: Shipping information. This may be the address of the client, or a mailing list.
- **selling offer**: created by a subscribed printer, identifying the service location, service type and capabilities.

Job preparation

containers

The first point of contact with the client is the creation of a shared document representing the job. The user's experience of this may be through a traditional web-interface that allows them to upload their artwork. The way we activate documents is to place them inside a *container*. A container is a web-service, able to converse with clients through document exchange, supporting basic message handling functions. We add the new job by sending it to the container in a *message* that defines the initial state of the job.

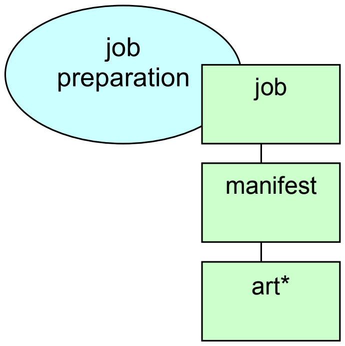


Figure 4: job preparation

The job, as illustrated in Figure 4, defines the job manifest, including links to the relevant artwork. The state of the job is represented as an XML document fragment. We think of the container as the root node of a single large document containing many business objects including the job. The container reacts to messages that define new objects by adding them to its children; the new job becomes *part* of the container. So that we can refer back to the job, a unique identifier[i] is added. The complete state of the container, including the new job, can be serialized to XML. An XML fragment illustrating the flavour of this containment is shown below.

```
<container>
      <job id="myjob">
            <manifest>
                  <art>mydoc.ps</art>
                  <art>myimg.jpg</art>
            </manifest>
      </job>
</container>
```

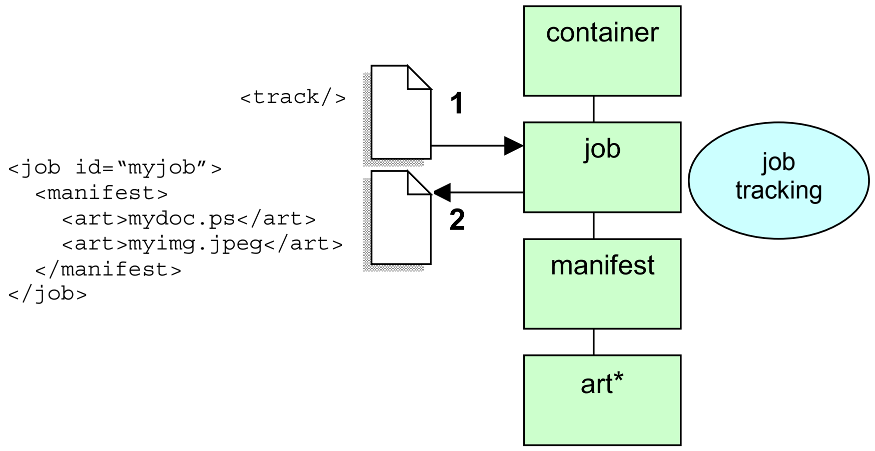
[i] Following usual XML identifier conventions, this name is a unique to the container; we don't need to know exactly where it is and it may even be moved within the containing document.

Job tracking
messages, components and actions

The second requirement is the ability to be able to view the job for tracking purposes. We wish to be able to request a simple *snapshot* of the current state. The component model is message based, and a snapshot is obtained by sending a request and waiting for the response. The creation of the job required a message to be sent to the outer container, but in this case we need to directly address the part of the document that we wish to view. A container exposes its parts as web-services. The job was named as we saw in the previous section, but this means that it has a distinct identity as a web accessible resource. We also want it to be able to perform some action on demand. While containers support a few primitive operations that provide basic access to the state, more complex actions are carried out by components associated with the part. For any given request of a part, its behaviour is delegated to the appropriate component. We send the request directly to the job, not to the component that actually does the work. The approach is document-centric, decoupling the client from the functional component.

Assume we can use a *track* message to request a copy of the job. The role of a container as message handler is simply to direct incoming messages to the appropriate functional components. The action accompanying the message is performed by a component associated with the receiving part and message type, as determined by their main element names ('job' and 'track'). The general behaviour of message-handlers is as follows. They are initiated by external input; the receipt of a message. As far as the container is concerned actions are atomic, with a response being generated on completion. The actual work is performed by components, web-services designed to work with document-state. The component is selected according to the message type and the type of part to which it is addressed.

From Figure 2 we can see that job tracking can begin at any time following job preparation. This means that we should be able to perform job tracking concurrently with other activities. In relation to the container, actions carried out by components are performed *atomically*; changes made by the component are not visible to others until it has completed. The view it has of the state represents a *consistent* snapshot of that state at the time it was invoked. The component enjoys the fiction that it is running alone, in *isolation* from other processes. Finally, we'd like some guarantee that when the process is complete, its results are available in as *durable* and persistent a fashion as can reasonably be achieved. The reader will recognise these as the ACID properties of transactions.



```
<track/>
```

```
<job id="myjob">
  <manifest>
    <art>mydoc.ps</art>
    <art>myimg.jpeg</art>
  </manifest>
</job>
```

Figure 5: job tracking

In Figure 5, the *track* message is addressed to the job (1), which invokes a job tracking component to perform the required action. The response (2) is the serialized state of the job (not including the surrounding container).

The job tracking component realizes a simple function of the current state. In response to other messages we may wish to insert information into the state (just like the constructor of the previous section). Yet another kind of request is one that polls the state for outgoing messages, allowing clients to pull messages from a part.

Job specification

dynamic type

Although job preparation doesn't communicate directly with other components, through shared state it has established the right conditions for follow-up actions such as job specification. Unlike job preparation and tracking, job specification is not triggered by input messages, but by changes to the state (the appearance of a job). A pre-condition of job specification is simply the existence of a new job within the container. If the pre-conditions are met then a job specification component is located that will run pre-flight tests on the supplied artwork. If the job checks out then it appends a detailed job specification suitable for production. This specification will include additional information about the finishing options and timescale required by the client. These changes may be summarised as post-conditions of this action. In practice, we need provide only sufficient detail for automated tools to discover the linkage between this activity and the next.

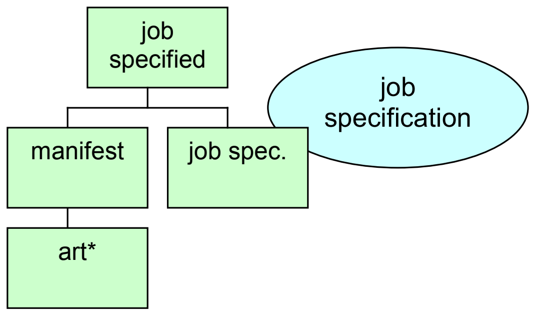


Figure 6: job specification

The association of documents with behaviour is not new[13]. Compound documents aggregate content from different applications, in effect composing content from different sources. In the web component model, different parts of a document are associated with different component services. The concept of document state provides us with a powerful model for XML processing. Every action must end with the serialization of all relevant state back to the container. If we ignore the very real side-effects of actions, we can look on them as functions over documents. From an information perspective we can focus purely on the effects of these actions on the document state.

Strongly typed programming languages don't like you messing with the types of objects once they've been created; it gives the compiler a hard time. Web components are more forgiving. The type of a part is given by its element name, so the type of `<job>` is 'job'. To signify the change to a fully specified job, Figure 6 changes this to `<job_specified>`, effectively changing the type. The impact of this is significant, as the part will now only use behaviour associated with 'job_specified', not 'job'. For one thing this provides a way to stop the job specification operation being re-triggered all over again, as it no longer applies to parts of this type. Dynamic types allow us to partition the number of operations we need to consider at any one time.

Dynamic types can be put to good effect to classify the different states that a part may pass through during its lifetime. We can describe the job lifecycle with the finite-state diagram of Figure 7. The state diagram abstracts away the fine details of any particular state, ignoring the details of the documents that each of these states represent. A more realistic analysis of print jobs would consider additional states such as error conditions or re-prints, and we would derive a more complex non-linear structure.

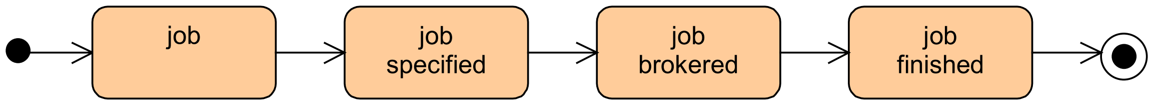


Figure 7: part lifecycle

Broking

activities and access control

Broking is not simply the invocation of a remote computational procedure. Even by itself, broking represents a rich collaborative activity. All parties who are involved in this activity are privy to the transactional context, so besides the broking component this includes the client and subscribed printers. Unlike simple actions, state is shared between these parties, though still not with anyone outside this context.

A print broker takes the job spec and turns it into a standard format suitable for inviting quotes. The Request For Quotes (RFQ) is a buying offer, designed to be compatible with selling offers put up by subscribed printers. Where the match between the RFQ (buying offer) and selling offer looks promising, the broker will notify the subscriber who may then (after some time) submit a quote. Printers submit quotes simply by sending the appropriate messages to add them to the RFQ, which is used to collate the responses. When the bidding period has ended the client will select one of the quotes, assigning the job to one of the printers. The RFQ may indicate how long the bidding period will last. Figure 8 shows the state of the job at the end of this process. Information that is irrelevant to the broking process is greyed out.

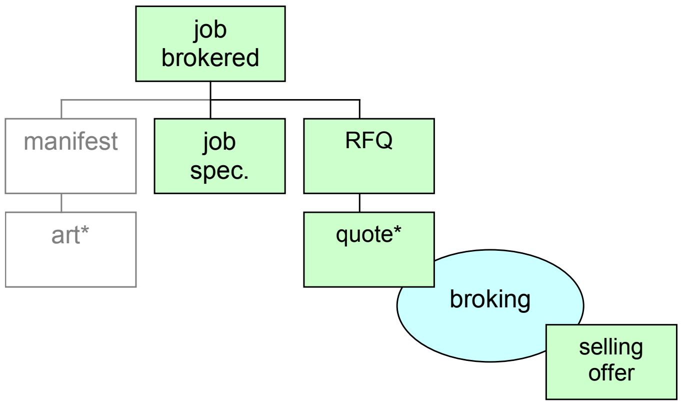


Figure 8: broking

Earlier, we discussed the transactional properties of actions with respect to document state. In that case the component only needed to obtain a lock on a part and its (strictly aggregated) sub-parts. In this case, the selling offer is not part of the job, as can be seen in Figure 8. It may be in the same container as the job, in different containers, or may simply be a document on the web. In the latter case, such documents are assumed to be constant, and components are unable to modify them. The other cases are more interesting; the state of the job together with the candidate selling offers represents a business transaction.

Business transactions are unlike database or distributed object transactions. A traditional database transaction is a logical unit of work that applies to a set of tightly coupled entities in a persistent store. Distributed object transactions extend this idea, permitting the objects to be physically distributed. They also provide support for transactional processes as long as they support the full complement of ACID properties. However, distributed transactions usually take place within a common environment, giving the transaction controller full control over the objects that fall within the transaction context. Furthermore, both traditional and distributed transaction models assume ownership of the data; an assumption that breaks with cross-enterprise business processes. Business transactions must work with loosely coupled objects (both logically and physically) and should not assume ownership of resources, nor control over the environment[14].

Use-cases involving multiple actors (see broking in Figure 1) have much in common with communication-based process modelling developed by Winograd and Flores[15]. Rather than modelling work done, they focus on modelling the interactions and commitments between participants. The Winograd/Flores *conversation for action* model is also customer centric in that it focuses on customer value and satisfaction. Communication-based processes range from loose ad-hoc collaborations to well-defined interactions such as the broking process. They define the notion of the *workflow loop* that represents a conversation between the customer (the client) and a performer (the printer). The workflow loop is taken to be the elementary building block for business processes. A collaboration may have internal state, but this would be lost if it were to be prematurely cut off. Only at the end of this process when an agreement is reached, do we need to commit this to the container.

Access control is a necessity. Just because the job is a web-based entity does not mean that everyone has equal rights to view or change it. The broking process would be unfair if bidders were able to see quotes already submitted by their competitors. Rights associated with the actor (role-based access) allow printers (subscribers) to read the RFQ and read/write their own quotes, but nobody else's. Actors are instantiated dynamically as part of the ongoing process. Subscribers are given the right to access the RFQ via the broker, acting with the authority of the client. Only these subscribed printers have the right to submit a quote.

Production & delivery

long-running transactions and concurrency control

Print is very much a "clicks and mortar" business where information services are blended with physical production and transport. Transactions that involve real people doing real work, such as production & delivery, tend to be much longer lived. For this activity to be successful it must have some guarantees about the stability of the information it depends upon, as highlighted in Figure 9; this information must be locked down. However, these *long-running transactions* carry a number of implications. Failure of the transaction becomes more likely with extended duration. Offers may be regarded as provisional, and any participant may back out of the transaction at any point. One way to ameliorate this problem is to relax the notion of transaction atomicity, an approach taken by the Oasis Business Transaction Protocol[16] (BTP). We may allow multiple successful outcomes, by which a transaction may succeed even if some (non-critical) operation fails. If failure is inevitable, and we have to undo the effects of a long-running transaction, *compensating* mechanisms[17] may be used to cancel changes already made, where strict roll-back is impossible.

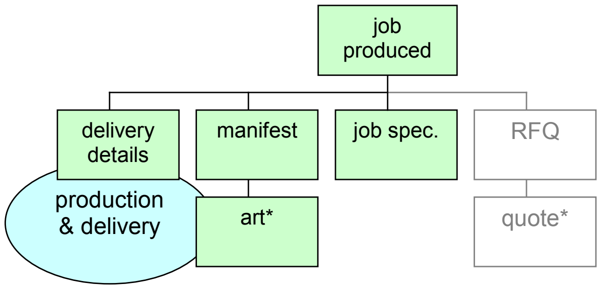


Figure 9: production & delivery

Aside from the problems of coping with the failure of long-running transactions, we must provide flexible concurrency control mechanisms that support concurrent access. While the broking activity allows shared access to participants within the same transactional context, we may also wish to make changes visible to external parties. For example, we may wish to expose some of the stages within the production process to job tracking. Many locking schemes distinguish between types of lock, such as reads and writes. Concurrent reads may coincide, whereas mixed reads and writes may leave the object in a confused state. These kinds of mechanisms allow the client to perform a *dirty read* of the job state even while the production & delivery operation is still active. However, even writes will mix. Predicate/ transform locking[18] was designed as a concurrency control mechanism for so-called long-running units of work. It introduces the idea of using a predicate as a condition for entering the commit phase. This permits concurrent writes to the same object, as long as their conditions are met. Working with the pre- and post-conditions of an activity, and ensuring they are compatible, buys us quite fine control over concurrent access.

Summary

We have proposed a web component model that emphasises the separation of state from behaviour. The novelty of this approach arises out of moving this architecture out onto the web, so that both state and behaviour become separately addressable web resources. The advantages of such an approach are apparent in web based collaborations where we can decouple service components from each other, relying instead on shared state.

---

[1] David Bovet, Joseph Martha, *Value Nets: breaking the supply chain to unlock hidden profits*, Wiley, 2000.
[2] William M.Adams et al, *Collaborative Commerce: The agile virtual Enterprise Model*, http://www.agileweb.com/pdf/ch12.pdf
[3] Stan Davis, Christopher Meyer, *BLUR: the speed of change in the connected economy*, Capstone, 1998.
[4] D. Heimbigner, The Process Wall: A Process State Server Approach to Process Programming, http://www.cs.colorado.edu/serl/process/Wall.html
[5] Rohit Khare, Adam Rifkin, Capturing the State of Distributed Systems with XML, http://www.cs.caltech.edu/~adam/papers/xml/xml-for-archiving.html
[6] Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001, http://www.w3.org/TR/wsdl
[7] Frank Gilbane, fourth forum XML, Paris 2001
[8] D. Georgakopoulos, M. Hornick, An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure.
[9] Rational, UML resource center, http://www.rational.com/uml
[10] Ivar Jacobsen, *The object advantage: business process re-engineering with object technology*, Addison Wesley.
[11] OPEN: Object-oriented Process, Environment and Notation, http://www.open.org.au
[12] Doug Rosenberg, Kendall Scott, *Use Case Driven Object Modelling with UML*, Addison Wesley, 1999.
[13] http://www.cs.berkeley.edu/~wilensky/CS294/lectures/OpenDoc.pdf
[14] Jim Webber, Mark Little, Savas Parastatidis, (www.arjuna.com) "Making web services work", Application Development Advisor, Nov/Dec 2001, vol.5, no.9, http://www.appdevadvisor.co.uk/Downloads/ADA5.9pdfs/LettersfromFront5.9.pdf
[15] T. Winograd, R. Flores, *Understanding Computers and Cognition*, Addison-Wesley, 1987.
[16] http://www.oasis-open.org/committees/business-transactions
[17] XLANG: *Web Services for Business Process Design,* http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm
[18] B.Bennet et al, A distributed object framework to offer transactional support for long-running business processes, 2000, http://www.research.ibm.com/AEM/lruow.html