Architecture and Design of an XML Application Platform

Russell Perry, Peter Rodgers, Royston Sellman
Digital Media Systems Laboratory
HP Laboratories Bristol
HPL-2004-23
February 19, 2004*

XML, XSL-T, workflow

A host of standards are emerging for the processing of XML. Examples of these are XSL-T, XInclude, XQuery and XML Encrypt. Whilst individually useful, there is not yet a framework to assist developers in the composition of such operations into more complex processes. In this report, we introduce the XML Application Platform (XAP), which aims to address this need. The XAP is shown to have wide application to any domain where XML processing is encountered. We also describe a research prototype we have built called Dexter. Dexter provides execution of XML workflows to achieve complex XML transformations whilst providing many low-level support features such as caching and resource pooling. Several examples are shown to introduce readers to typical usages and to illustrate the workflow language that was developed for creating applications within Dexter.

Contents

1 Introduction

In this paper we discuss the design of an XML application run-time or *platform*. Our interest in this topic was sparked by the observation that flexible and efficient manipulation of XML documents is of critical importance, not just to the bulk transport, but also the end point processing of data for many types of distributed application. The work began in the summer of 2000 when we produced a report on Electronic Business XML (ebXML) [1], and developed an implementation of IFX [2], an XML-based protocol for the exchange of financial data and instructions over the Internet. The IFX prototype allowed user to view utility bills on both a PC and mobile phone using XSL-T files to transform user data (stored in IFX) to either XHTML or WML. These activities demonstrated to us that whatever the application - web-services, an eCommerce protocol, or a dynamic web-site – there was frequently a need to create and manipulate XML documents, and that this creation/manipulation layer could be abstracted out into a common platform. The syntactic and semantic mechanisms used to request these XML documents and to indicate whether they are intended for human or machine consumption is not significant at this level; these can be considered application level issues. For example, a machine producing XHTML or IFX need have no semantic understanding of the structure or interpretation of the XML.

This observation led us to contemplate the design of a common platform on which a range of XML applications could be developed. We envisaged an XML Application Platform (XAP) which, under control of an application, would be capable of ingesting XML from external web services, internal enterprise applications or local storage and which would manipulate the data and return new processed XML documents to the calling application. As such, the XAP would represent a sound foundation upon which to implement high-level XML applications such as ebXML, RosettaNet [13] BPEL4WS (which supersedes earlier process language efforts XLang and WSFL) [14], or even function as a web server.

We saw that this vision of an XAP was being made more feasible by the growing number of conventional applications and web services that deliver XML either natively or through adaptors. However, we also observed that although many open standards for the *manipulation* of XML exist or are planned (see Table 1), there is a lack of similar technology for easily orchestrating such operations and thus producing XML applications. Without such a technology the developer is left to manage an application strewn over both code and XML documents. Worse still, if the developer uses many of the programming-like constructs available in languages like XSL-T, the application logic rapidly becomes unmanageable. Thus a hoped-for consequence of developing an XAP will be to provide the developer with a framework for creating applications using XSL-T and other transforms in a manageable and productive way whilst allowing them to be integrated with additional code. Ultimately XML transforms can be reused in the same way as software modules or packages.

From the outset it was clear that the XAP would need to be extensible, so that existing and future point solutions could be integrated as pluggable modules. We also made it a goal that developers should be able to create applications using a simple-to-author workflow language, which would be executed by the platform. A workflow would specify sequences of document-centric operations, which when executed would result in the construction of a XML *response* document to return to the calling application. Ultimately, we decided that the language should also be pluggable, rather than fixed on a single proprietary dialect, which would risk alienating potential developers.

Table 1: Examples of point solutions for XML manipulation

Type	**Examples**
Access	XPath [3], XPointer, XLink [4]
Query	XMLQuery [5]
Transformation	XSL-T [6], XSL-FO [8]
Signature, Encryption	XML-Sig [9]
Validation	Xschema [10], RelaxNG [11]
Editing	XUpdate [12]

Whilst meeting the requirements described above, we envisaged that the XAP could also be used to tackle many typical lower-level performance issues. For example, enabling the parallel execution and caching of pre-computed XML transformations could reduce high latencies in web service calls. In addition optimised in-memory representations of the transforms could be cached by the XAP to increase speed of subsequent transformations.

The outcome of the work was the development of an XAP prototype called *Dexter*. Initially a first XAP prototype was completed in autumn 2001, which helped refine and make concrete, some of the early thinking, and was the predecessor to the Dexter prototype described in this report. This report presents an overview of Dexter, describing the core principles of its design and the high level functional units developed as part of the implementation. The next section describes the historical context of web applications within which we believe the XAP to be the next step. Section 3 elaborates on the key design principles that underpin the architecture. Sections 4 and 5 contain more detailed description of Dexter's architecture, its main functional units and entity classes. Section 6 contains several examples, which have been developed to test Dexter and illustrate its usage and which serve as an introduction to the idoc workflow language. Section 7 is concerned with the future possibilities for XAPs in the context of other leading edge web developments. Finally, section 8 wraps the report up with a short conclusion.

2 XML Processing, Web-Services and Beyond

XML is everywhere. The qualifier "XML-enabled" is now attached to all kinds of IT product, from relational databases to web browsers, office application suites and smart phones. This, and the fact that XML is such a simple idea – structure a text document with (potentially) unique text-based tags - make it tempting to imagine XML is a done deal and that there is little or no research left to be done. We intend to show that this is not the case. To do this, we offer both historical and future-looking perspectives on Internet technologies and on relatively newer web-services initiatives. Our aim is to show that an XML Application Platform is a natural next step in the suite of Internet technologies required to support the development of web services and eCommerce.

2.1 The Evolution of Web Sites and Web Servers

Historically, Information Technology has evolved by adding functionality to core systems and then building applications to exploit the new functions. In turn, many of the applications are tools that accelerate the addition of new capabilities. Eventually a point is often reached where the additional functionality is found to be so useful that it is implemented across all platforms. It then becomes part of a new *layer*, which application designers can take for granted while they concentrate on higher-level abstractions. The key observation is that a symbiosis exists between layers and the application tools that are built on them (and with which they are built). An improvement in one benefits the other. In Figure 1, we have attempted to capture this evolutionary path for the set of Web-related Internet technologies. This has been separated out along several dimensions.

The lower part of Figure 1 shows the gradual separation of the three kinds of data that characterise web applications: presentation specific (i.e. layout, style etc.), content specific, and business specific. Many commentators have noted that this separation makes good engineering sense but it has taken several years to come about. It is wise to keep this time scale in mind when considering the development of the web; not everything happens at breakneck speed. With the arrival of the first web browsers in late 1993 and 1994 a step change in the use of the Internet was triggered. Principally this took the form of an explosion of web sites, which were browsed by users running freely downloaded client software. Initially, web sites served static HTML pages with the only technical twist being simple page caching at both server and client ends. Very quickly though, the introduction of server side programming resulted in developers adding dynamic content to web pages. Generally this took the form of small snippets of program code embedded into static page templates. The code usually held specific application logic but also acted as a gateway to common external systems, such as databases.

Over time the support for this type of programming led to the so-called 3-tier architecture which consisted of relatively sophisticated web servers serving dynamic web pages generated by "middleware" logic drawing on resources stored in databases. Initially the middleware logic was usually connected to the web-server via the standard Common Gateway Interface or CGI [15], and implemented in a host of different languages: C, C++, Tcl, Perl and others were popular choices.

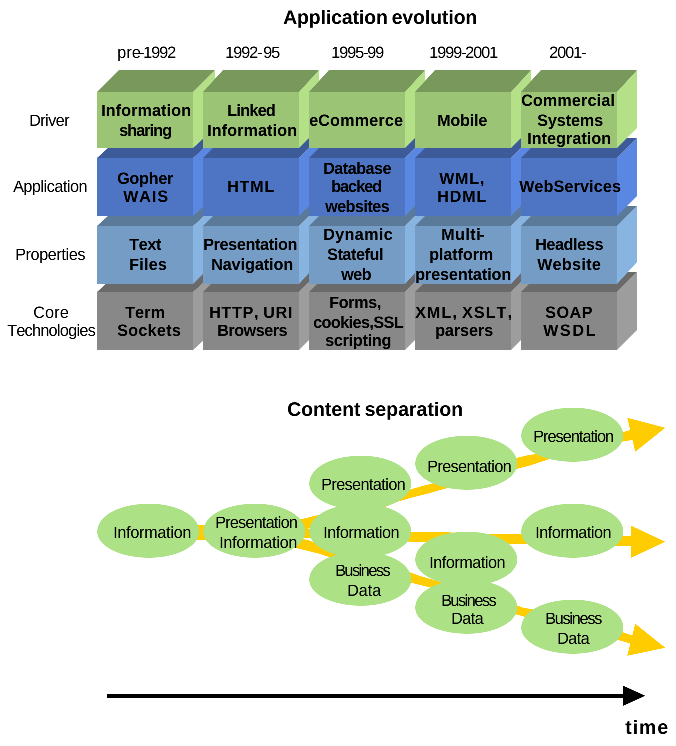


Figure 1: The Evolution of Internet technology layers

Later on, two other important technologies for middleware's web server interface were introduced: Java Servlets/Server Pages and Microsoft's Active Server Pages, and these are now the dominant players. In a refinement of this architecture the all important (and expensive) database connections are often pooled so that they can be shared among processes as and when required.. In addition, many companies now offer consolidated solutions to the complex problem of building and maintaining dynamic web sites. Such solutions (for example Vignette StoryServer) support the production of static and dynamic HTML pages and contain libraries of reusable components that handle common tasks such as content management, layout design, and centralized control of look and feel. Also, a number of companies sell web *portal* systems, which provide a common web interface to new and legacy enterprise applications. Some of these consolidated solutions are built on generic 3-tier systems, others are self-contained, but the end result is similar from the user and operator perspective. Web applications have developed a distinct look and feel compared to desktop applications, reflecting the nature of the medium.

2.1.1 The move to XML

As noted above a long-standing difficulty in the web-publishing process has been the mixing of logic, presentation and content data within a single dynamic document. An early response to this issue was the introduction of Cascading Style Sheets (CSS) in 1996, which made possible the independent control of layout and style for documents in Web browsers. However, CSS is a point solution and does not help in the common scenario where content data is being dynamically generated from a database or another program. Nor does CSS help with the issue of supplying markup to the multiplicity of devices (not only PCs but also mobile phones, PDAs, kiosks, consoles, Set-top-boxes etc) connected to the networks. Finally, CSS does not enable clean engineering of the *return channel* – i.e. the method (usually an HTML *form*) by which data is passed back from user to application. An increasingly popular solution to these problems is the storage of content data in one set of XML documents and presentation data in another. These can then be worked on independently and *combined* to produce target markup when needed. XML Stylesheet Language (XSL) is one global standard for this representation, and XSL Transformations (XSL-T) together with XSL Formatting Objects (XSL-FO) have become the standard technologies for doing the combination work.

2.1.2 Adding client-side functionality to web pages

One way of making pages dynamic involves embedding server side objects into a web page. The server detects these embedded objects at request time, executes them and inserts the resultant fragment in place of the object. An alternative model is to embed an object in the returned web page that is then executed by a browser. This model, typified by JavaScript, Java applets, and browser plugins such as Macromedia's Flash, allows dynamic behaviour to be offloaded to the users browser.

Recently, a new attempt to support client side behaviour has been started, which builds on the principles of XML. The XForms [18] standard attempts to support the creation of dynamic forms using pure XML. XForms adopts a model-view-controller design pattern where the data model, rules and binding to GUI components are all declared in XML. The aim is to enable a client to process an XForm document and construct an appropriate GUI to input and validate data. Recently, a project with similar ambitions been initiated by Microsoft called XDocs [16].

We believe this type of approach is significant because it represents logic, content data and visual information in XML. This allows different parts of an application to exist on and be processed by different machines, and allows us to imagine applications that operate (conceptually at least) entirely in the XML domain.

2.2 Emergence of Web Services

The most recent step in IT deployment is the Web Service paradigm, which relies on XML for the transport of data and invocation of functionality. XML has been widely adopted beyond its early use as a device independent language. Viewed as a new incarnation of distributed computing the web service approach is less principled than previous efforts, but is a pragmatic response to the risks and benefits of Internet based communication. Whereas previous examples of distributed computing have tried to provide interoperability through globally standard interfaces and exchange protocols, web services (behind the hype) have a goal of achieving interoperability through the discovery, negotiation and establishment of data exchange formats and protocols on the fly. The idea is that providers can publish content or services in an XML dialect (consisting of vocabularies and protocols) that service-consumers can discover, read, and convert to a form that is suitable for further processing in their systems. Much consideration has been given to the publishing and discovery of *schemas*, which attach type information and a form of meaning to data in XML documents. However even when all the XML dialects involved are standard, and are understood by all parties, core services will rely upon XML transformation. For example, content will be built by fragment integration from disparate data-sources, structural information will need to take account of the status of the parties in a transaction (whether they are machines or human, how they have been authorised, how much state they can hold and so on) and presentation data will need to be generated for a range of different fixed and mobile client devices. There will have to be facilities for encrypting any or all of this data, and for distributing the authorisations and keys.

2.3 Towards an XML Application Platform

We believe the widespread adoption of XML will drive the requirement for networked clients to support at least some basic level of XML processing. We believe that the adoption of XML in web services and other domains has not yet been matched by the development of technologies to support XML domain processing. In keeping with our comments in section 2.1, we envisage that XML processing capability will, over time, become an integral part of the Internet/Intranet infrastructure.

As listed in table 1, there are many types of operations that can be performed on an XML document. Composing several of these operations to complete a task is an obvious future need, which is not currently well supported. We envisage that there is significant value in developing a common runtime to support this composition process, which we have called an XML Application Platform (XAP).

Because adopting XML across the board means that content, logic and presentation components are all authored in a common format, this brings within reach a higher level of abstraction that can be exploited to construct such a common runtime. Furthermore, it ensures a large range of application domains for an XAP to be deployed.

A high-level layered view illustrating the positioning of an XAP within the existing infrastructure is shown in Figure 2. In this view, the XAP supports a range of transport protocols and XML macro operations such as XML Query. Above these layers, an XML workflow language allows a developer to define arbitrarily complex compositions of operations, which the XAP can execute. Applications thus interact with the XAP by requesting the XAP to execute a workflow and waiting for the response document. An obvious example of this could be a web server requesting execution of a workflow that generates a web page.

In the model shown in Figure 2, the XML workflow layer is not a conventional workflow engine, but is a lightweight engine focussed on sequencing and composition of XML operations such as XQuery and XSL-T. Just as general workflow engines are “programmed” with workflow documents, a XML workflow language is similarly required for execution by the XAP. However, because all operands are formatted in XML, it is appropriate to define a new workflow language, appropriate for XML domain processing. This is very distinctive from typical programming paradigms such as JSP/ASP, which embed snippets of code inside markup. The intent is to

- Simplify development of XML rich applications, whilst avoiding embedded code inside mark-up.
- Enable parallel execution of XML operations where possible to tackle latency e.g. external web service calls when aggregated can introduce significant latency.
- Provide a framework where XSL-T, XQuery and other macro operations can be reused between applications i.e. encourage modularity of XML rich applications.

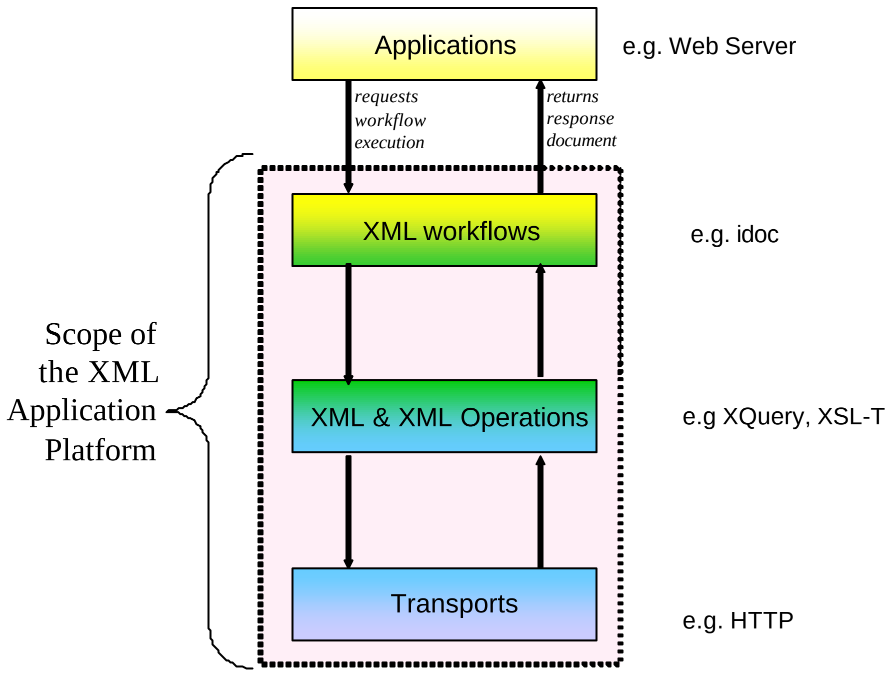


Figure 2: Conceptual model for the XML Application Platform

Furthermore by authoring the workflow language in XML it is possible to

- Simplify workflow development by building GUI tools to edit the workflow document
- Execute a workflow to dynamically generate a new workflow, which is then itself executed.
- Manage all workflows, content and presentation data in a similar manner i.e. all entities in the XAP are homogeneous.

With these benefits in mind we now move on to a discussion of some principles for the design of an XAP which we have arrived at through both practical experience and more theoretical considerations.

3 Design Principles for an XML Application Platform

This section introduces a set of design principles for the XML application platform. These principles were derived both from analysis of the evolving technological substrate of XML and from our experiences in building two prototype systems. The first prototype system was demonstrated in April 2000), and at that time supported the construction of web pages from a set of workflow documents defining page composition. The second prototype was constructed during 2001 and included a pair of tools for authoring workflows and managing document links in an XML-based web site application.

3.1 Everything is an XML document

Under an XAP, all information, meaning all instructions, data and node state, is held in XML documents. Facing outwards from the XAP, serialized XML documents are used to carry information across network connections and to persist information in long-term storage. The XAP is thus responsible for managing the lifecycle of XML documents.

3.2 Processing in the XML Domain

A key influence has been the recognition that whilst the use of XML for data interoperability has grown rapidly, the mechanisms for *processing* that data frequently demand that the developer drop out of the XML domain to use APIs such as JAXP, MSXML, JDOM and DOM4J. The manipulation of XML in the programmatic domain is independent of the web-infrastructure, leading to the unhealthy mixing of transport, message, presentational, and business logic code. Minimizing the requirement to move out of the XML domain makes it easier to provide a declarative framework that can adapt to schema changes and simplifies the work of developers who have to extend code in response to changes in schemas.

3.3 XML Operators

We should explain here what an *operator* is in the XAP context. In current XML terminology, document transformation has become synonymous with XSL-T [6]. From the XAP point of view XSL-T represents a useful set of operators for expressing certain kinds of XML transforms. In an XAP an operator controls the conversion of one XML document data structure into another. This definition embraces XSL-T, but allows for other transform sets such as XUpdate, XEncrypt or XACML [21], and null transforms such as schema validation, or even the custom conversion of one XML dialect into another.

3.4 Extensibility

However, not all processes map comfortably into the declarative style and an XAP should not seek to implement *all* middleware business logic in the XML domain. An XAP should support extensibility by allowing developers to create new operator plug-ins, which encapsulate lower level logic. These operators can be called upon by the XAP workflow engine to perform macro level operations.

3.5 Support for XML Standards

To ensure the XAP has a sound conceptual basis we have drawn on a range of important XML standardisation efforts. Primarily we have looked to the work of the W3C, which has formulated (or is reformulating) several canonical operation types (such as XUpdate and XEncrypt mentioned above and, even more crucially, XPath) in terms of basic operations on a strictly defined XML document called the XML Infoset [22].

3.6 Document Lifecycle Management

It should be possible to exert fine-grained control over the lifecycle of the XML documents managed by the XAP. Examples of policies to control are caching, persistence and access control.

3.7 Scalability

The wide range of possible applications for an XAP requires that it is scalable both in performance and functionality. Functionality is supported by the extensibility principle, but also by allowing the workflow language itself to be modified. To enable high performance, parallel workflow execution to reduce latencies and efficient caching of data and previously computed operations should be possible.

3.8 Network Transports

It should be possible for clients and applications to communicate using the best transport protocol for the job. Sometimes it is useful to receive a response on different transport than was used for requesting execution of a workflow. The XAP supplies an abstraction layer for transports to achieve these goals.

3.9 Ease Of Use

This is the last, but an important overarching principle. If the workflow language is not easy to use then it will represent a barrier to adoption. By authoring workflows in terms of XML domain operations and keeping the instruction set minimalist the development and debugging process is simplified. GUI development tools are also more easily supported, but this is not to say that such tools are required to hide complexity.

4 Key Abstractions

In this section we describe the key conceptual abstractions we have developed for our version of an XAP architecture, which we call Dexter. The implementation and interfaces that realise these abstractions are presented in section 5.

4.1 XMLBeans

Although XML document parsers and the W3C Document Object Model are widely established, they are more concerned with bringing XML documents into the programmatic domain than managing them as resources in their own right. In Dexter, recalling principle 3.1, XML documents are *the* central resource and so the ability to apply resource locking (i.e. read/write locking), caching and so forth requires additional interfaces. To support this functionality, an XMLBean class was developed which encapsulates an XML document and implements the interfaces required to manage the document.

Also, many applications require that different documents can be combined in some sense. However, this requires that we finesse the problematic merging of XML trees. XMLBeans, therefore, provide this functionality and allow documents using different schemas to be logically grouped and managed together without having to merge their internal Infosets [22]. XMLBeans are a fundamental concept within Dexter, and in section 5.2 we will describe them (and some important sub-classes) in much more detail.

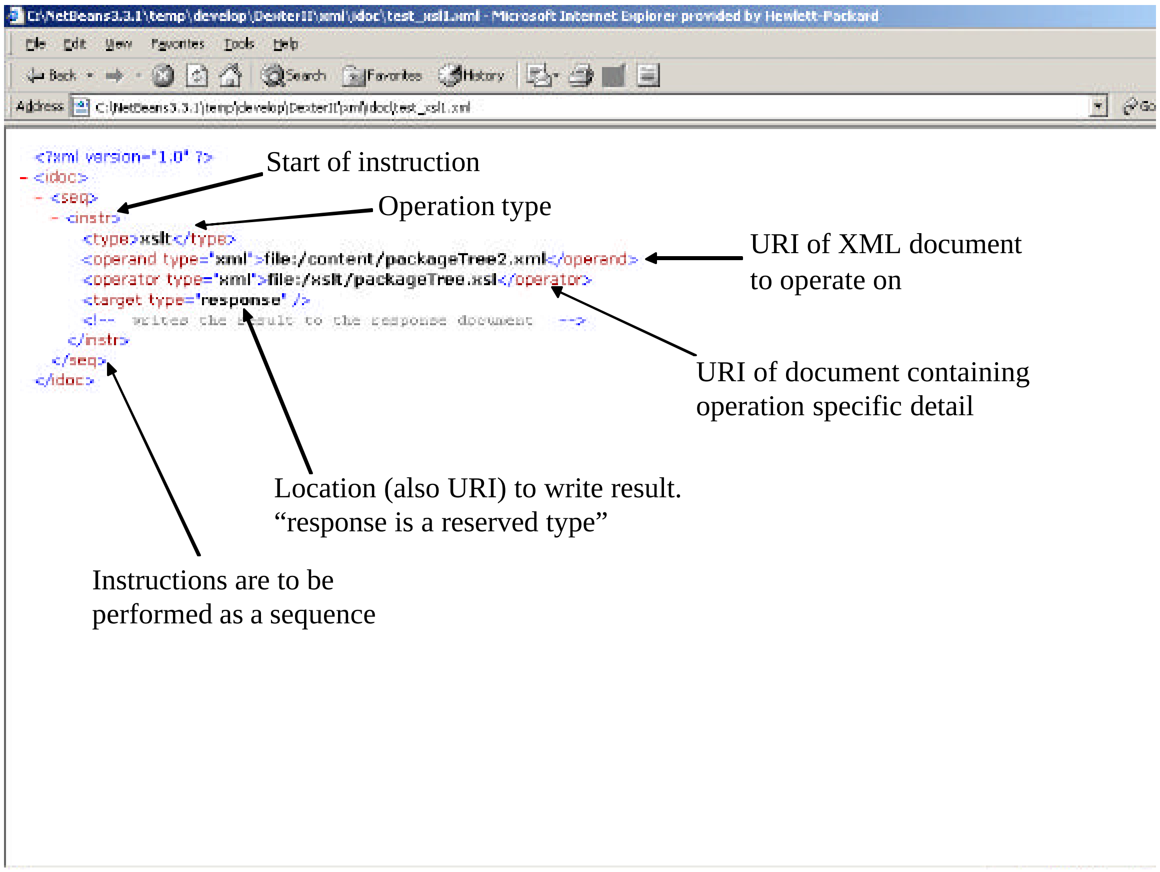
4.2 A Container for XMLBeans

As is the case for Enterprise Java Beans, a *container* is defined to support and manage the lifecycle of the XMLBeans. The container is responsible for instantiating, caching and persisting XMLBeans as well as performing workflow execution. Rather then managing all XMLBeans alike, the lifecycle of an XMLBean is managed according to a defined set of policies that can be applied to a group or individual XMLBeans. This is a much finer level of granularity than can be applied to Enterprise Java Beans.

4.3 Execution Languages and the IExecutable Interface

A workflow language refers to the language for specifying the sequence of operations to be performed on or by specified XML resources. All XML resources are identified by URIs. Early on, it was decided that the choice of workflow language (and associated interpreter) should be pluggable. To support this, the IExecutable interface was defined. The IExecutable interface is used by the Container to control the execution of the workflow, and any interpreter that implements IExecutable can be plugged into Dexter. To enable a developer to get started quickly, a default workflow language referred to as the “idoc language” has been defined. The idoc language is only summarised in this document, but several illustrative examples are given in section 6, which should be more immediately helpful than a dry schema document.

To make our discussion of idocs more concrete we introduce a simple example here. The annotated idoc is shown in Figure 3 and the XHTML response document produced by Dexter is shown later in Figure 4. The idoc simply requests that an XSL-T transformation is applied to a static XML file and the result returned to the requestor.


C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\test_xsl1.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\test_xsl1.xml
Go
Start of instruction
Operation type
URI of XML document to operate on
URI of document containing operation specific detail
Location (also URI) to write result. “response is a reserved type”
Instructions are to be performed as a sequence

```
  <?xml version="1.0" ?>
- <idoc>
  - <seq>
    - <instr>
        <type>xslt</type>
        <operand type="xml">file:/content/packageTree2.xml</operand>
        <operator type="xml">file:/xslt/packageTree.xsl</operator>
        <target type="response" />
        <!--  writes the result to the response document  -->
      </instr>
    </seq>
  </idoc>
```

Figure 3 : A simple idoc workflow illustrating an XSL-T transformation

The top-level root element is labelled as *idoc*, but this has no special significance. The child element *seq* instructs the workflow engine to execute each instruction sequentially, although this does *not* mean that an instruction must be *completed* before another is started. This idoc contains a single instruction defined under the *instr* element. The *type* element specifies what type of operator to apply, in this case an XSL-T transform. The *operand* element specifies the XML source document that is to be transformed, and the *operator* element specifies the document that defines how the operand is to be transformed. Both the operand and operator also contain the attribute *type* set to ‘xml’, which somewhat confusingly indicates that the documents are stored locally as files. Finally the *target* element defines the location to write the result of the XSL-T transformation. In this case the *type* attribute is set to *response*, which is a reserved keyword. The response is the name given to the document that the idoc constructs by its execution and which is sent back to the requestor upon completion.

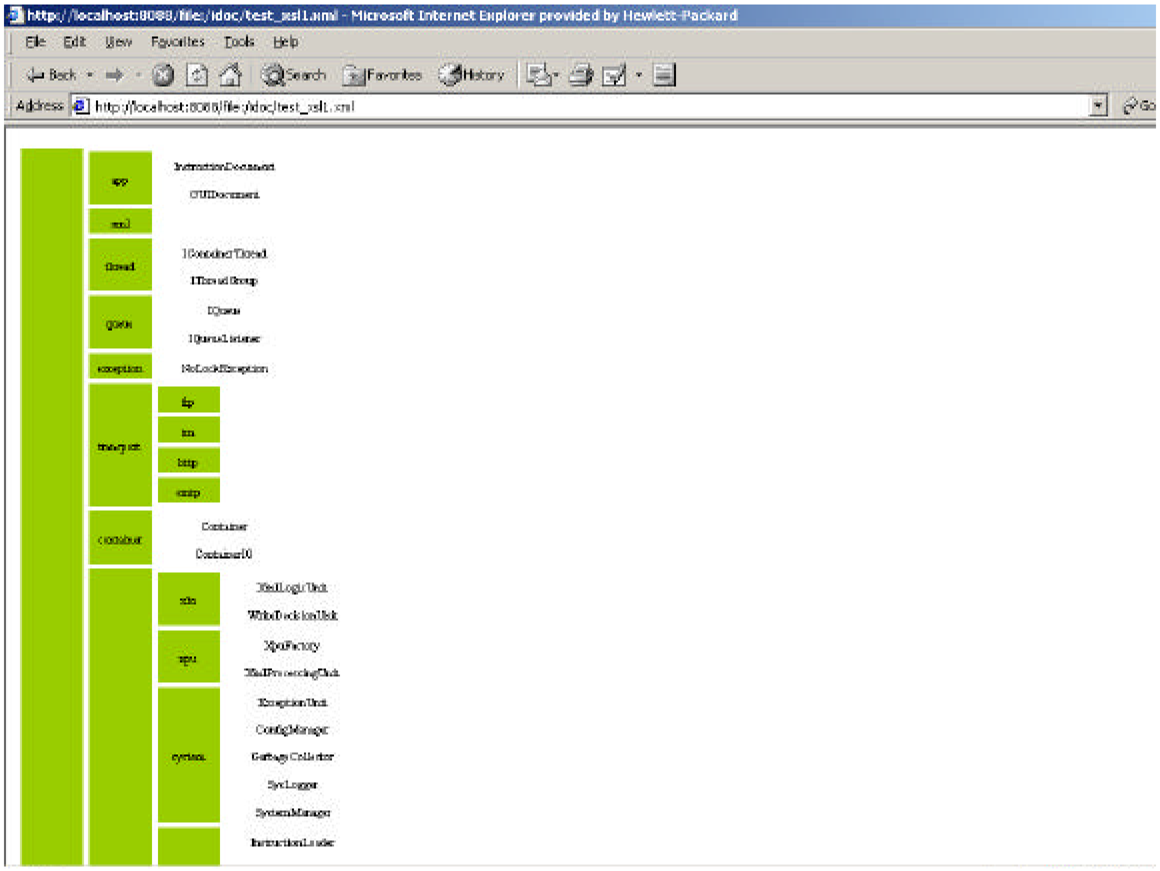


Figure 4 : Part view of the java source tree generated by executing the idoc shown in Figure 3

4.4 Document Accessors and Active URIs

To support the "*Everything is a Document*" principle, it is necessary to abstract all resources as XML. In many cases, such as a web service call, or an XML data file this is natural, whereas for a relational database conversion from a data set to an XML schema will be required.

Document Accessors are Dexter's resource abstraction method. However, they do more than just parse an XML source or convert a non-XML resource into XML. Accessors are also responsible for persisting an XML document to its long term storage medium should it be declared to require this behaviour. Many Document Accessors also provide implementations of workflow operators such as XQuery and XSL-T.

The Document Accessors developed to date can be classified into two main types, *Transport Accessors* and *Active Accessors*. Each type of accessor returns an XML document, which the XAP uses to instantiate an XMLBean.

Transport accessors are required to provide support for IO operations such as file or http requests. Active accessors are somewhat more complex and are best explained with reference to the example idoc in Figure 3. Essentially an active accessor performs an operation on an XML source document. So in the example above an XMLBean encapsulating an XSL-T document implements an additional interface, *IDocAccessor*, to support requests for transformation (see Appendix B for details). Additionally, it is necessary to be able to uniquely name the resulting document. To provide for this, an *Active URI* scheme was created which comprises the URIs of the operand and operator documents and the operator *type*. For this example the active URI (key words in bold) is:

active:xslt**@**file:/xslt/packageTree.xsl**+src@**file:/content/packageTree2.xml

Thus a document with the active URI shown above represents the result of applying the XSL-T transformation defined in packageTree.xsl to the source document packageTree2.xml. Conversely a request for a document with the active URI shown above is tantamount to requesting the XSL-T transformation of packageTree2.xml using packageTree.xsl. In many cases accessors are also XMLBeans and thus inherit all the useful behaviours described in section 5.2 below. Generally, however, we refer to XMLBeans that expose functionality through the IDocAccessor interface as accessors.

5 Dexter Internals and Execution Flow

This section contains a description of Dexter's top-level entities. Although there is effectively only one entity class (excluding the container), namely XMLBean, the set of Document Accessors represents such an important and significant subclass that we discuss them separately.

5.1 The Container

As its name implies, the Container is used to manage the lifecycle of all the other system components as well as XMLBeans. These components are introduced in section 5.4. When the container is started, all the components are started and the container is then able to service requests through the transport layer. The container also performs the function of routing internal messages between system components.

5.2 XMLBean Entities

There are several technologies in existence which aim to make life easier for programmers writing code which must process XML data. Examples in the Java world include JDOM [http://www.jdom.org/], the Java API for XML Binding (JAXB) [http://java.sun.com/xml/jaxb/], and, recently, BEA's XML Beans [http://www.bea.com/events/webinars/xmlbeans/index.shtml]. These technologies facilitate a useful binding between Java objects and XML data, and are particularly useful for loading, validating and writing XML. They also assist in simple node-by-node navigation through the associated DOM.

Dexter's XMLBean system goes much further. It is a fully fledged architecture which not only facilitates the binding between XML and implementation language but also enables powerful XPath-like navigation to nodes or sets of nodes inside the document. The XMLBean system also provides a supporting environment to deal with life-cycle and resource management issues typical of complex objects in distributed systems that may come under heavy loads. Appendix B shows the Java documentation for the most important parts of the XMLBean system.

All Dexter entity classes use the IXMLBean interface. XMLBean implements this interface and is the base class of most entity classes in Dexter. There is one other class that directly implements IXMLBean. This is called *JellyBean*, and it was created to avoid representing all *data* internally as documents. The IXMLBean interface is itself made up of five sub-interfaces to support management of the entity beans:

- **ILockable**
    - Provides an interface for acquiring and releasing a lock on an XMLBean. This interface returns classes implementing the **IReadable** and **IWritable** interfaces, which allow an XMLBean document to be modified.
- **IPod**
    - Allows a bean to logically group other beans (documents). An XMLBean can thus refer to other beans. Each reference to an XMLBean is given a unique name.

- **ICacheable**
    - Supports managing a bean within a cache.
- **IPassivatable**
    - For beans that can be passivated and activated from the passivation *store*.
- **ISafeXMLBean**
    - A mechanism for ensuring a thread safe access to a bean by a document accessor.

Details of the interfaces can be found in Appendix B.

Within Dexter there are several document types that are used to exchange information between system components. To handle some of these document types, sub-classes of XMLBean have been created which provide specialised interfaces to allow more efficient interaction with the underlying XML Document. Table 2 shows the mapping between each document type and the XMLBean subclass that encapsulates it. Not all documents have specialised XMLBean sub-classes.

Table 2. XML Document to Dexter XMLBean Entity mapping

Document Type	**Associated XMLBean Subclass**	**Purpose**
MetaDocument	MetaBean	Groups together the policy documents which specify how the related XML document is to be managed
Idoc	RunnerBean	This is the interpreter for the idoc workflow document
Execution Context Document	ContextBean	Groups together several XMLbeans relating to the execution of a workflow
Response Document	XMLBean	A standard XMLBean, which contains the response document to send to the requesting client.
Exception Document	ExceptionBean	Created to hold information about an Exception
Parameter Document	XMLBean	An XMLBean holding parameters associated with an idoc workflow

Expanded summaries describing each document type and the associated XMLBean sub-class are given in the following sections. To orientate the reader, the relationships between the various documents are shown in Figure 5. The ContextBean is the principal object that is passed between system components during a workflow execution and is generally the top level document in the logical tree of related documents. The relationships between the principle XMLBeans shown in table 2 is provided by the IPod interface. So for example in Figure 5, the ContextBean maintains direct references to four other beans namely, the ContextMetaBean, the RunnerBean, the Response Bean and the Param Bean.

5.2.1 ContextBean

The *ContextBean* is short for *Execution Context Bean*. A unique ContextBean is created for every workflow execution request. Principally, the ContextBean provides a grouping (using the *IPod* interface) of all the documents related to the execution of an idoc. It contains references to the idoc (RunnerBean), the Response Document, the parameter document and an Exception document (ExceptionBean) should an exception be raised during execution. It also provides the means by which the response document is returned to the correct requesting client.

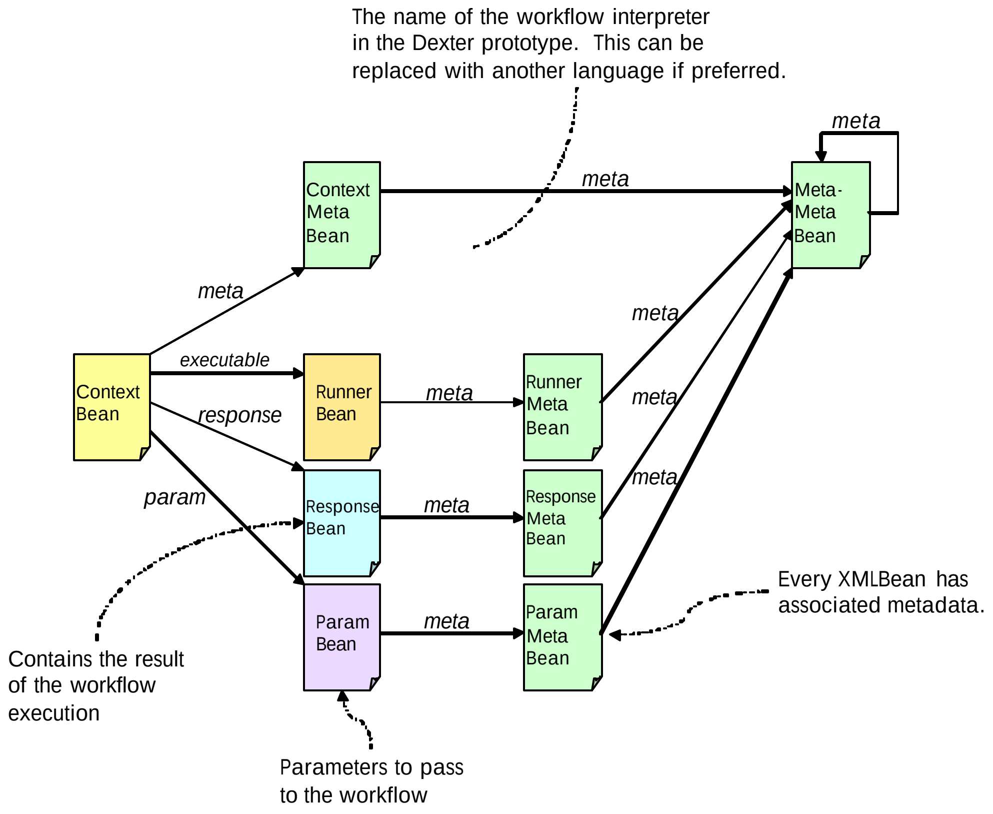

The name of the workflow interpreter in the Dexter prototype. This can be replaced with another language if preferred.
meta
Context Meta Bean
meta
Meta-Meta Bean
meta
meta
executable
Context Bean
Runner Bean
meta
Runner Meta Bean
meta
response
param
Response Bean
meta
Response Meta Bean
meta
Every XMLBean has associated metadata.
Param Bean
meta
Param Meta Bean
Contains the result of the workflow execution
Parameters to pass to the workflow

Figure 5: Relationships between the document types used to perform execution of an idoc.

5.2.2 RunnerBean (class implementing the IExecutable Interface)

This is the class that encapsulates an idoc, instantiates the idoc language interpreter, and manages the execution state for all clients that may be concurrently executing the same idoc. It is, therefore, a relatively complex XMLBean subclass. The RunnerBean implements the IExecutable interface, which describes how the Container determines which instruction to perform and what operands are required at each step of the program. The Container executes instructions blindly, as it is not aware of the client or idoc for which it is performing an instruction. Effectively the container sees a sequence of XML domain instructions to be performed. At each iteration of the IP-ID cycle, the RunnerBean interprets the idoc and determines the next instruction to perform. Thus the actual implementation of the IExecutable interface encapsulates the evaluation of conditional flow control or other familiar constructions that might be used in a workflow language. Each method of the IExecutable interface contains a ContextBean parameter in order that the RunnerBean can associate a request with a particular execution context. The IExecutable interface is shown in more detail in Appendix B.

In section 3.4, a stated principle was to allow developers to extend the XAP to provide additional functionality or to provide alternative implementations of workflow control. To achieve the latter, the developer must provide an interpreter for their new workflow language, which implements the IExecutable interface. So, for example, a workflow could be modelled as a state transition machine. Going further, a specialised class could be created that does not require the use of a workflow document. Also, because all executable classes implement the IExecutable interface, Dexter can support the execution of multiple different workflow languages at the same time. So particular functionality can be provided using an authoring language most naturally suited to the task in hand.

5.2.3 Response Document

Inside Dexter, XML domain operations are defined to accept an input XML document or fragment and output a document or fragment. Viewed from outside, Dexter operates by accepting a request for the execution of an idoc and returning a document called the response document. For example, if Dexter, in executing an idoc, is acting as a web server, then the response document would be a web page.

Thus *response document* is the general name given to the document that will ultimately be returned to the client. Internally the response document is held as an XMLBean referenced from within the ContextBean. The response bean is often referenced as the copy target for an XML operation as would be required, for instance, when building a web page from a set of fragments.

5.2.4 Parameter Document

It is possible for parameters to be passed into an idoc by use of the parameter document. The ParamBean, which is simply an XMLBean, encapsulates the parameter document. There is a reserved keyword "param" used to reference a parameter document within an idoc workflow. The parameter document only has local scope within a specific execution context. Examples of its use are provided in section 6. All parameter beans have the root element name *param*.

5.2.5 MetaBean

A MetaBean wraps an XML Infoset that describes the details of the XMLBean with which it is associated. The various aspects of how an XMLBean should be managed are grouped together in its related MetaBean. This allows XMLBeans to be given behaviours analogous to, but richer than those of Enterprise Java Beans in J2EE. Some basic behaviours such as persistence, statefulness, and session affinity policies are available as defaults in the system – for example the RunnerBean always has a persistent behaviour described in its MetaDocument – but other behaviours are easily added. It is a requirement in Dexter that all XMLBeans should contain a reference to a MetaBean. A consequence of this is that all MetaBeans are assigned to a standard MetaBean called the *MetaMetaBean*. To prevent infinite regress, the MetaMetaBean is its own MetaBean.

The same MetaBean can be applied to all XMLBeans of a certain type, or can be assigned individually. This assignment is performed by a rules engine, which is configured by a meta-control document. In our current implementation, the MetaBean can optionally contain references to additional documents controlling different facets of the document lifecycle, e.g. a *cache policy* document and an *access control policy* document or it can contain information internally. Currently only a cache policy, which provides details of how an XMLBean should be cached, has been implemented.

5.2.6 ExceptionBean

Whenever a system component encounters an exception during an operation, an ExceptionBean is created and added to the ContextBean. The ExceptionBean contains information concerning the error, and can be used by other Dexter components to handle the error condition gracefully. Generally the ContextBean, containing the ExceptionBean is placed on the ExceptionUnit unit (see section 5.4) for error handling. For example, an ExceptionBean document can be passed directly back to the client or could be formatted using an XSL-T transformation.

5.3 Document Accessors

As stated in section 4.4, there are two main classes of document accessor, transport related and active. An active accessor encapsulates functionality and is the class responsible for implementing the different types of XML operator type. In many cases, the operator is defined by an XML document (e.g. XSL-T), but this is not a requirement. As such, active accessors are the mechanism by which developers can extend Dexter's functionality. Tables 3 and 4 below describe the accessors written to date, grouped according to whether they are transport or active.

Table 3. Listing of the Transport Accessors

Protocol	**Accessor Name**	**Notes**
HTTP	HTTPAccessor	The HTTP Accessor accesses documents stored on a remote web server or simple Web services.
Fork / exec (Dexter)	InternalAccessor	Requests execution of a nested workflow i.e. one workflow requesting execution of another. The resulting response document is returned by the accessor.
File	FileAccessor	The File Accessor is useful for Beans which represent data on the filesystem local to Dexter. A typical usage is configuration beans. The root of File URIs is not the root of the local filesystem, but an offset configured per Dexter instance. It is defined in the file *dexter.properties* in the jre/lib directory of the VM running Dexter. This contains one line: basepath=<local path>
Null	NullDocumentAccessor	The Null Accessor is useful for Beans which have no initial state and are never persisted. Beans which fall into this category are Document Accessors and transient stateful beans.
Literal	LiteralAccessor	XML documents can be specified inside idocs (i.e. inline) by specifying a literal type. The LiteralAccessor is used to access these XMLBeans.
LDAP	LDAPAccessor	Uses the LDAP to perform a query.
Introspection	SystemComponentAccessor	Allows any of the functional units to be serialised to XML.

Table 4. Listing of the Active Accessors

Standard / Project	Accessor Name	Notes
RelaxNG	RelaxNGAccessor	This Document Accessor performs RelaxNG Schema Validation. It uses the Jing RelaxNG package. It returns a Boolean bean indicating success or failure.
Part of Dexter	PackageTreeAccessor	Generates an XML representation of the Dexter source tree.
Apache XML-security	XsignAccessor	Performs XML Signing. It uses the Apache XML-security package for crypto. Documents are signed with the Dexter instance private key held in the [Dexter root]/security/.keystore Java keystore.
Apache XML-security	XverifyAccesor	This Document Accessor performs XML Signature verification. It uses the Apache XML-security package for crypto.
XQuery	XqueryAccessor	Performs an XQuery on the input XML document. Uses XCool [29].
XSLT	XSLTAccessor	Accessor to wrap XSLT and apply transform to XML input document. Assumes assumed will be used with the same operator value each time around. So the transform is cached. Uses Xalan [30].
XChange (part of Dexter)	XChangeAccessor	This Document Accessor performs XChange operations on an XML document (i.e. document edits)
XPath	XPathEvaluator	Evaluates an XPath expression on a document. This returns a BooleanBean which is a simple XMLBean containing either true or false. This is primarily used for control loops.

5.4 Functional Overview

The Dexter architecture is based around a container which manages all other entities. For prototype and first release Dexter has been implemented in Java but Dexter is principally an instance of an architecture for a platform to process XML applications and could be as easily implemented in C# on .Net as Java on a JVM.

An overview of the main functional components is shown in Figure 6. To describe the components we follow the steps taken to execute an idoc (or any other workflow). The principal steps are shown in Figure 6 using blue numbered squares. Each of the main components - the UniversalIO, Universal Server, Instruction Pump, Instruction Decoder and DocumentIO are all based on a common super class which provides a message queue and a user defined number of threads to process incoming messages. Each of the main components then implements logic specific to its role. The superclasses for these components are not described in this report since they are developed from relatively standard design patterns. Messages from one component to another are sent via the container, which handles the routing.

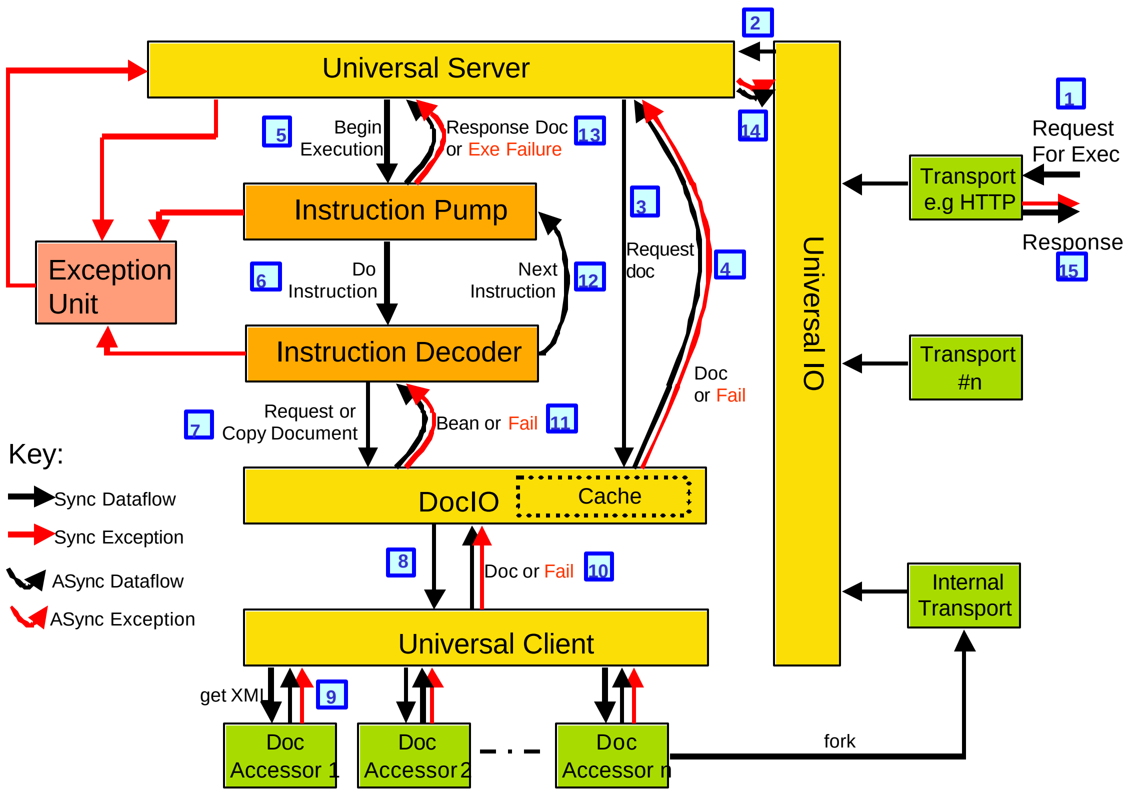


Figure 6. Major functional units of Dexter and execution flow for a workflow

5.4.1 UniversalIO

This execution process begins with Dexter receiving a *request for execution* (RFE) for an idoc (step 1). The RFE can specify either a URI referencing an idoc contained in the Dexter source directory, or it may actually contain an idoc for execution (i.e. pass by value). There is no defined scheme for mapping an RFE on to a protocol or a transport layer, since it was felt that this process should be left open to the developer. To provide flexibility, the UIO supports a pluggable transport manager within which common transports such as SOAP, HTTP Servlet, Java RMI, FTP and SMTP can be augmented with additional domain specific transports like MQSeries and, Java Messaging Service. The developer may therefore choose the manner in which client applications communicate with the Dexter XAP. Furthermore while requests and responses generally will be handled via a single transport, it is possible for the UIO to support asymmetric routing of responses. For example, a processing request might arrive on an HTTP Servlet channel and the response could (after a satisfactory HTTP closing response) be issued as a MIME attached email over SMTP. In the current implementation, only the HTTP and Internal Transports have been developed.

Whilst the UniversalIO (UIO) is a shown as a part of the Dexter container, ultimately, we envisage that the container will support a single interface for receiving execution requests and the transport specific plug-ins will be deployed as part of other infrastructure components that wish to delegate XML processing to Dexter.

The UIO passes the idoc, any associated parameters and transport context information to the Universal Server, which, as its name suggests, coordinates the generation of responses to the incoming requests (step 2). In our current implementation, the parameters are encoded in a simple XML document consisting of name value pairs. The mapping is very simple and, therefore, also fairly generic. The root element is by convention named *param*. Each HTTP parameter is mapped to a child element of the same name, and each child element so created contains the parameter's value.

5.4.2 Universal Server

As described in section 4.1, all documents within the system are encapsulated by XMLBeans. The DocumentIO component (DocIO) is responsible for instantiating these XMLBeans and thus the US must communicate with the DocIO to obtain the set of XMLBeans required to perform the execution of the idoc. So in step 3, the US requests a new Execution Context Bean, the RunnerBean (in the case of the idoc language), a Response Bean (just an XMLBean) and an optional parameter bean, also an XMLBean, to hold the parameters sent in the RFE.

These document requests result in the DocIO returning a ContextBean, a RunnerBean and a ResponseBean (Parameter Bean) to the US (step 4). Note that the execution context bean could be indexed by an application specific session identifier, obtained indirectly or established in a prior interaction with the processor. The session identifier can then be used to maintain execution state across multiple sequences of request-responses in a long running interaction.

Upon receipt of all three (or four) XMLBeans, the US logically groups them by adding the RunnerBean and ResponseBeans (and Parameter Bean) as child beans of the ContextBean (as in Figure 5) using the IPod interface. Note that the DocIO automatically adds MetaBeans to each XMLBean.

5.4.3 DocumentIO

The DocIO module hides the details of accessing and parsing documents from the requesting component. It provides access to resources that are remotely hosted on external file systems, databases, web services, or on other Dexter systems. It achieves this by delegating request to an appropriate Document Accessor (implementing the lower level transport protocols). The DocIO contains multiple threads (a configurable number) to enable multiple requests to be made in parallel. The DocIO contains a cache, which holds previously accessed documents (encapsulated as XMLBeans). This enhances performance by avoiding costly external requests repeatedly if the results are already available. The policies for managing the documents inside the cache are determined by the attached MetaBeans. A controller thread, inside the cache, performs garbage collection of XMLBeans that are stale and no longer referenced by any other component in the system. The functioning of the cache is fairly complex and is not described in this document for purposes of readability.

For reasons of efficiency, Dexter makes frequent use of *references* to XMLBeans. However, without careful design, the Cache could remove XMLBeans that are still being referenced. To guard against this, both short and long term *memory fixing* is implemented. For example, the UIO must place a long fix on the ContextBean to guarantee that it remains in memory until the associated RunnerBean has completed execution. Memory fixing in the Cache is another complex topic that is omitted here for reasons of readability.

5.4.4 The Instruction Pump and the Instruction Decoder

Continuing with the execution flow, the ContextBean is now passed from the US to the instruction pump (step 5). This unit uses the IPod interface of the ContextBean to obtain the references to the executable RunnerBean. The Instruction Pump (IP) drives execution by requesting the RunnerBean to provide its next instruction. In fact, the RunnerBean maintains a separate instruction pointer for each execution context it is operating in, which allows it to service multiple sessions. If an instruction is available the IP passes a reference to the ContextBean to the Instruction Decoder (step 6). If there are no more instructions, the ContextBean is returned to the US (step 13).

The Instruction Decoder (ID) now interacts with the DocIO to perform execution of the current instruction. This involves several interactions with the DocIO, internally defined by a state machine. Briefly, the ID first requests the operator, operand, parameter (if applicable) and target documents (step 7). Note that each document is requested individually. Internally the DocIO checks to see if the document already exists in the cache and if not relays the document request to the relevant DocumentAccessor (steps 8 & 9). The UniversalClient in Figure 6 provides access to the appropriate DocumentAccessor by examination of the requested URI.

When all the requested documents have been received (through notification step 11) the ID places a write lock on the target bean. It also increments the fix count on each bean to indicate that it has a reference to each bean. It then requests the DocIO to return the document with the active document URI (step 7 again). This causes the DocIO unit to perform the actual XML operation through an active document accessor. The resulting XMLBean is returned to the ID through step 11. Finally the ID requests that the DocIO copies the active document into the target document (step 7). The DocIO then returns a notification to indicate completion of the copy command (step 11). The ID removes the write lock from the target bean and then unfixes all requested XMLBeans to indicate that it has finished.

Finally in step 12, the ID returns the ContextBean to the IP to begin execution of the next instruction. This repeats until all instructions in the idoc have been completed.

5.4.5 Returning the Response Document

Once the IP determines that all operations have been completed, it passes the ContextBean to the US (step 13). The US extracts the response document and returns it to the UIO for transmission back to the requesting client (step 14). Note that the response transport and response delivery address need not be the same as the requesting client and this information is specified as part of the RFE. Finally the appropriate transport adapter is given the response document to return to the client (step 15).

5.4.6 The Exception Unit

In the event of an exception occurring at any stage of this process, each of the units (US, IP, ID) will generate an ExceptionBean which is added as a child of the ContextBean. The ContextBean is then sent to the Exception Unit. The Exception Unit performs clean up functions and then sends the ContextBean to the US. The US checks to see if there is an ExceptionBean assigned to the ContextBean and will return that to the UIO in place of the response document. This feature is a great help in debugging an XML application – the raw XML in the ExceptionBean can be re-processed by Dexter into whatever form is most convenient: a web page, an email, or perhaps a WML message to a capable phone.

5.4.7 The Internal Transport

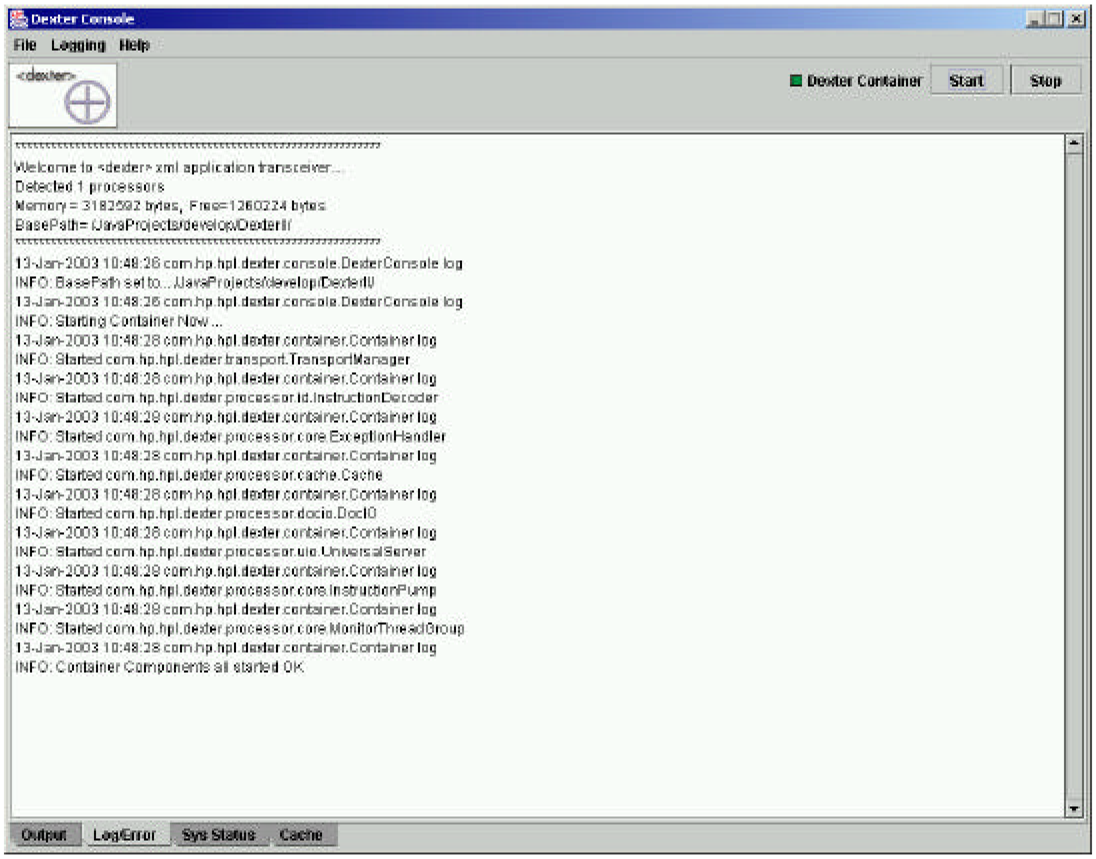
An important functional unit shown in Figure 6 is the InternalTransport. This transport module implements the same interface as any other transport adapter. However, its purpose is to support the execution of workflows by other workflows. Any component in the system wishing to execute an idoc simply requests the active URI with the workflow URI provided as the operand. Thus nested execution is performed as just another document request.

5.4.8 Support for Asynchronous Execution

In the IExecutable interface, the method *isAsynchronous* allows the Instruction Decoder to determine whether an instruction can be performed asynchronously. In the example above, the instruction is performed entirely synchronously, i.e. the execution for a specific context blocks while the relevant documents are accessed or computed. In asynchronous mode, the ContextBean can be returned to the IP to begin execution of the next instruction as soon as the target, operand and operator beans have been received, fixed and write locked if appropriate. Completion of the instruction continues as before, but in parallel with the next instruction(s). Because the target is write locked, any subsequent instructions requesting it will block until the asynchronous instruction releases the lock.

5.5 The Dexter Console

For the purpose of managing the XAP Container, a GUI was developed called the Dexter Console. A screen shot is shown in Figure 7. The console provides views of the container logging information.



```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>
Welcome to <dexter> xml application transceiver...
Detected 1 processors
Memory = 3182592 bytes,  Free=1260224 bytes
BasePath= /JavaProjects/develop/DexterII/
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>
13-Jan-2003 10:48:26 com.hp.hpl.dexter.console.DexterConsole log
INFO: BasePath set to.../JavaProjects/develop/DexterII/
13-Jan-2003 10:48:26 com.hp.hpl.dexter.console.DexterConsole log
INFO: Starting Container Now ...
13-Jan-2003 10:48:28 com.hp.hpl.dexter.container.Container log
INFO: Started com.hp.hpl.dexter.transport.TransportManager
13-Jan-2003 10:48:28 com.hp.hpl.dexter.container.Container log
INFO: Started com.hp.hpl.dexter.processor.id.InstructionDecoder
13-Jan-2003 10:48:29 com.hp.hpl.dexter.container.Container log
INFO: Started com.hp.hpl.dexter.processor.core.ExceptionHandler
13-Jan-2003 10:48:28 com.hp.hpl.dexter.container.Container log
INFO: Started com.hp.hpl.dexter.processor.cache.Cache
13-Jan-2003 10:48:28 com.hp.hpl.dexter.container.Container log
INFO: Started com.hp.hpl.dexter.processor.docio.DocIO
13-Jan-2003 10:48:28 com.hp.hpl.dexter.container.Container log
INFO: Started com.hp.hpl.dexter.processor.uio.UniversalServer
13-Jan-2003 10:48:29 com.hp.hpl.dexter.container.Container log
INFO: Started com.hp.hpl.dexter.processor.core.InstructionPump
13-Jan-2003 10:48:29 com.hp.hpl.dexter.container.Container log
INFO: Started com.hp.hpl.dexter.processor.core.MonitorThreadGroup
13-Jan-2003 10:48:28 com.hp.hpl.dexter.container.Container log
INFO: Container Components all started OK
```

Figure 7: Screen shot of the Dexter Console

6 Examples

In this section we describe several simple examples that have been developed to test and demonstrate the Dexter System. In order to be able to provide visual output, the examples all generate XHTML web pages. For each example, a description is first provided followed by an annotated view of the idoc and a screen shot of the output produced by Dexter from executing the idoc.

It should be borne in mind when reading the examples that schema validation of all documents used within the idocs can be performed using the appropriate accessors e.g. [11]. Document validation has not been shown, in order to keep the examples clear of too much clutter; to provide validation would only require the insertion of an extra instruction per validation. Also some examples are made somewhat more complicated than necessary in order to demonstrate different features of the idoc language or of Dexter itself.

All the file URIs as specified in the example idocs are *relative* paths from the Dexter root directory which is specified in the *dexter.properties* file.

6.1 Displaying the Dexter Source Tree

This is a very simple example to introduce the basics of the idoc language.

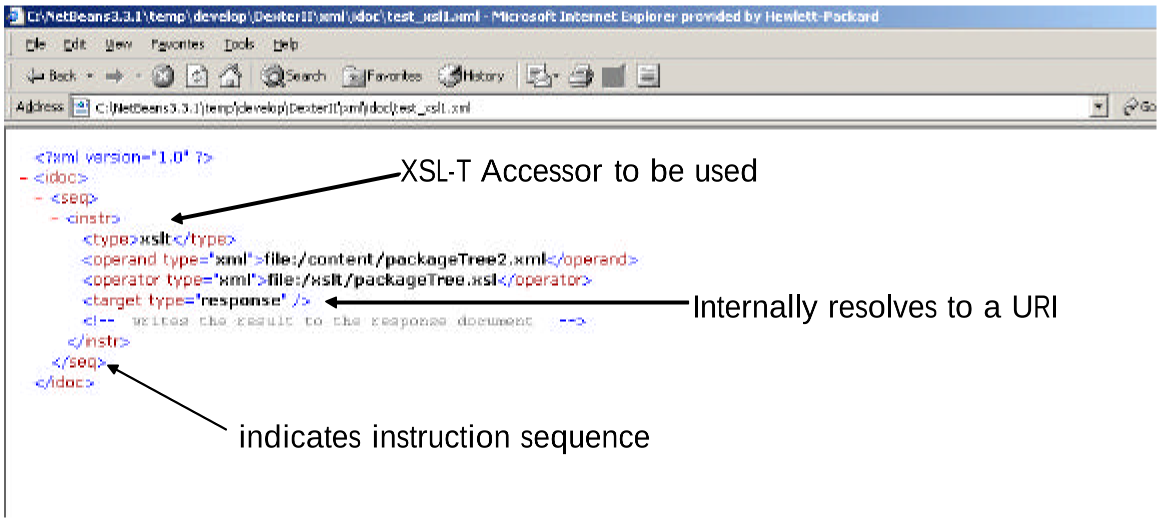
6.1.1 Description

This example illustrates one of the most common patterns in current XML practice: an XML transformation applied to an XML document listing the Dexter source code packages and class files. The idoc simply applies an XSL-T transform (packageTree.xsl) to an XML document (packageTree.xml).

6.1.2 The idoc

This idoc contains only a single instruction. The instruction *type* is specified to be XSL-T. Both the operand and operator documents are available on the local file system. The output of the transformation is written directly to the *response* document. The target element *type* attribute is set to *response* which is a reserved word within the idoc language. Internally the RunnerBean resolves the target to the URI of the unique response document created as part of the execution of this idoc (see section 5.4.2).

Although this example is very simple, it allows us to say something about the way a developer can extend Dexter with custom Accessors. For example, the operand document containing the source code listing was generated from separate code using a tree walker pattern. This code could itself be modularised as a Document Accessor and used as an operator. Similarly it is a simple matter to modify the idoc instruction to use a Javadoc Accessor to create documentation from the java source code. Furthermore it is possible even to compile the java code using an Ant DocumentAccessor.


C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\test_xsl1.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\test_xsl1.xml
Go
XSL-T Accessor to be used
Internally resolves to a URI
indicates instruction sequence

```
  <?xml version="1.0" ?>
- <idoc>
  - <seq>
    - <instr>
        <type>xslt</type>
        <operand type="xml">file:/content/packageTree2.xml</operand>
        <operator type="xml">file:/xslt/packageTree.xsl</operator>
        <target type="response" />
        <!-- writes the result to response document -->
      </instr>
    </seq>
  </idoc>
```

Figure 8: idoc to display the Dexter source tree

6.1.3 The screen shot

Only a part of the source tree can be shown in a single screen shot (Figure 9). In fact only the high level package names have been shown, excluding the leading com.hp.hpl.dexter package name. The complete source code listing is shown in stages in Appendix A using slight variants to the XSL-T style sheet to display different parts of the tree.

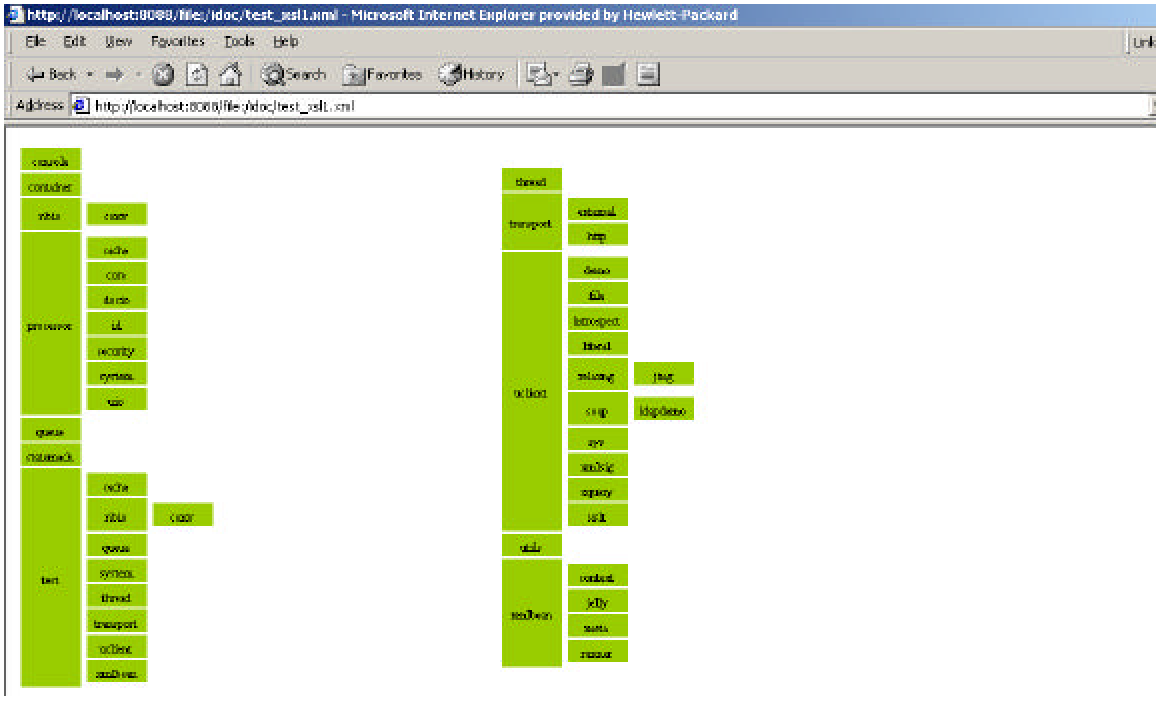

http://localhost:8088/file:/idoc/test_xsl1.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Links
Back Search Favorites History
Address http://localhost:8088/file:/idoc/test_xsl1.xml
console
container
processor
cache
com
id
security
system
queue
test
cache
queue
system
thread
transport
thread
transport
external
http
demo
file
introspect
thread
jpeg
soap
ldapdemo
xslt
utils
context
jelly
meta

Figure 9: View of the top level packages in Dexter source tree

6.2 Application of XSL-T and a Forked idoc

This is a more complex example, which shows how to nest the execution of idocs. This helps developers to modularise applications as sets of idocs.

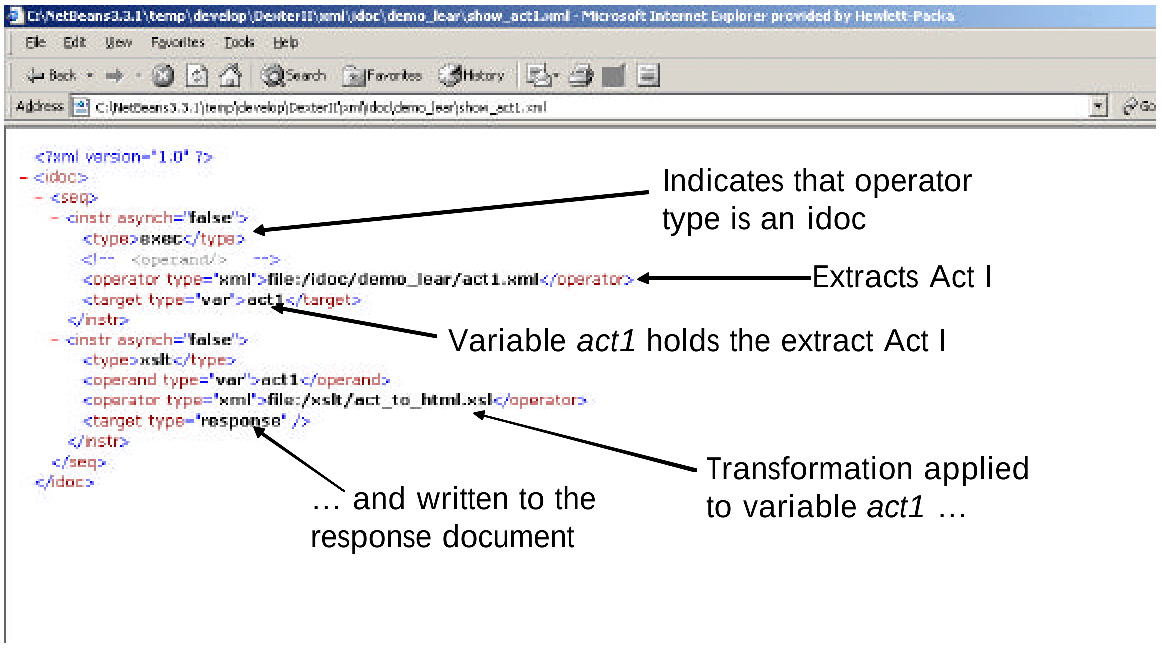
6.2.1 Description

In this example, Dexter is used to produce an XHTML page containing just the first act of Shakespeare's *King Lear*. The page is created in two steps, i.e. with two instructions, for the purposes of illustrating how to fork the execution of a subsidiary idoc during the execution of a primary idoc. In step one, the first act is extracted from the complete play and in step two the extract is formatted into XHMTL. The first instruction forks the execution of a second idoc, which just extracts the first act. The output is stored in a variable which is used in the next instruction. The second instruction applies an XSL-T transformation to the document contained in the variable thus producing the XHTML. Obviously in practise this could all be easily performed using a single XSL-T transformation.

6.2.2 The idoc

The idoc used to produce the XHTML page is shown below in Figure 8. The first instruction contains a request to execute another idoc (idoc.xml) and the second instruction applies an XSL-T transformation.

Note that the first instr element contains an attribute *asynch*. This optional attribute can be set to true or false. If set to true, the RunnerBean will be able to continue executing the next instruction before the first instruction is complete. This is helpful to allow concurrent access to external resources which involve significant latencies. Concurrency prevents the cumulative build-up of the latencies that can occur with purely sequential execution and which are particularly acute when the XML resources are not local to the machine hosting the container. In this example, because the second instruction is dependent on the availability of the variable *act1*, from the first instruction, it will block until the first stage is complete. This is true whether or not the first instruction is marked for asynchronous execution or not. Thus Dexter will ensure that parallel execution is exploited where possible, but will otherwise enforce a data driven execution flow i.e. operations are performed as soon as all the inputs are available.


C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_lear\show_act1.xml - Microsoft Internet Explorer provided by Hewlett-Packa
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_lear\show_act1.xml
Go

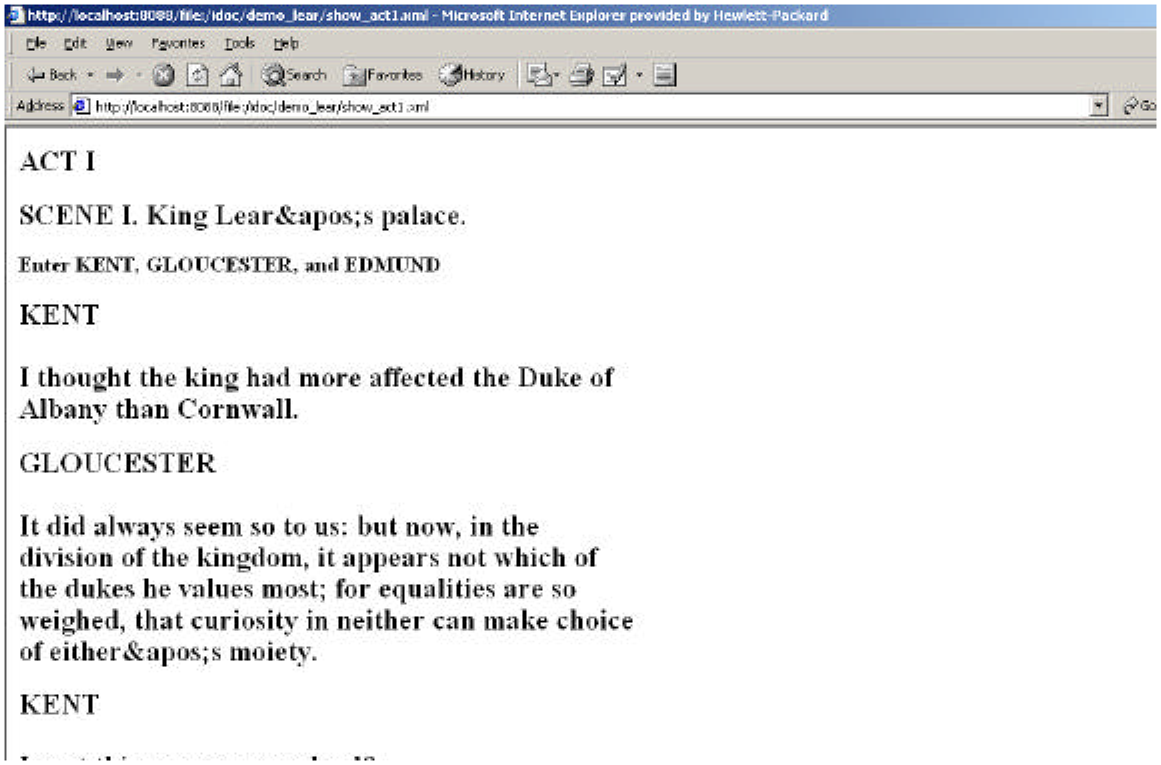
```
  <?xml version="1.0" ?>
- <idoc>
  - <seq>
    - <instr asynch="false">
        <type>exec</type>
        <!--  <operand/>   -->
        <operator type="xml">file:/idoc/demo_lear/act1.xml</operator>
        <target type="var">act1</target>
      </instr>
    - <instr asynch="false">
        <type>xslt</type>
        <operand type="var">act1</operand>
        <operator type="xml">file:/xslt/act_to_html.xsl</operator>
        <target type="response" />
      </instr>
    </seq>
  </idoc>
```

Indicates that operator type is an idoc
Extracts Act I
Variable *act1* holds the extract Act I
Transformation applied to variable *act1* …
… and written to the response document

Figure 10: The idoc to extract and display Act I from the play “King Lear”.

6.2.3 The screen shot

Figure 11 shows a screen shot of the output from executing the idoc. The URI shown in the browser address bar indicates the idoc which has been used to generate this page.


http://localhost:8088/file:/idoc/demo_lear/show_act1.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8088/file:/idoc/demo_lear/show_act1.xml
Go

ACT I

SCENE I. King Lear&apos;s palace.

Enter KENT, GLOUCESTER, and EDMUND

KENT

I thought the king had more affected the Duke of Albany than Cornwall.

GLOUCESTER

It did always seem so to us: but now, in the division of the kingdom, it appears not which of the dukes he values most; for equalities are so weighed, that curiosity in neither can make choice of either&apos;s moiety.

KENT

Figure 11: Extract of Act I from the play “King Lear”

6.3 Processing a form to view an idoc and using parameters

This illustrates how web form parameters can be used to send parameters to an idoc.

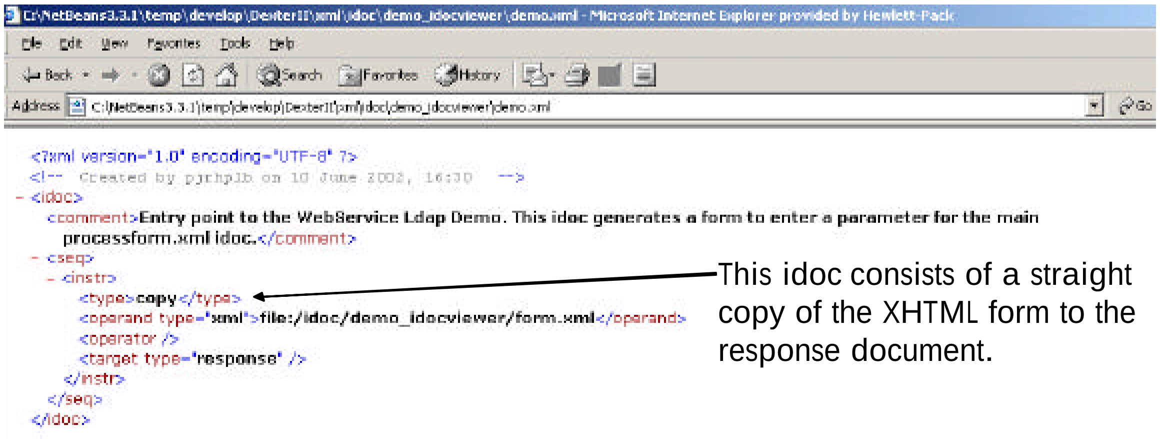
6.3.1 Description

This example consists of two interactions with the user through their browser. In the first interaction, an XHTML form is returned which allows the user to specify an idoc to view. The submitted form is then processed and sent to a second idoc for processing. This second idoc returns an XHTML page with the user-specified idoc presented inside a text edit window.

The example shows how parameters can be used inside an idoc. It also shows how the idoc can be used to fill in a template page, a familiar design pattern for building web pages. Finally it also introduces the *literal* type i.e. XML embedded inside an idoc rather than an externally referenced XML resource.

6.3.2 The first idoc (to display the form)

The idoc used to display the user form is very simple. It consists of a single instruction of type copy which allows the developer to simply copy (or more generally insert) an XML document into another document. In this case the target is the response document, but could equally well be a node within a document. The example illustrates how an idoc can be used to simply serve static web pages.


C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_idocviewer\demo.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_idocviewer\demo.xml
Go

```
  <?xml version="1.0" encoding="UTF-8" ?>
  <!--  Created by pjrhplb on 10 June 2002, 16:30   -->
- <idoc>
    <comment>Entry point to the WebService Ldap Demo. This idoc generates a form to enter a parameter for the main
      processform.xml idoc.</comment>
  - <seq>
    - <instr>
        <type>copy</type>
        <operand type="xml">file:/idoc/demo_idocviewer/form.xml</operand>
        <operator />
        <target type="response" />
      </instr>
    </seq>
  </idoc>
```

This idoc consists of a straight copy of the XHTML form to the response document.

Figure 12: idoc used for generating an XHTML web form (allows user to specify which idoc to view)

To complete the second part of the example i.e. the processing of the user's submitted form, an artificially convoluted process is described in order to bring out several idoc features and usage patterns. Principally these are nesting idocs and creating idocs on the fly and then executing them. A simpler approach is shown later.

Figure 13 shows a top-level view of how the response document is produced upon receiving the user's submitted form from part 1. This figure shows the three idocs/templates which are used to produce the response. Idoc_viewer.xml is the top level idoc to which the user submits the form for processing. The URI of the idoc to view, as specified in the form, is passed as a *parameter* to idoc_viewer.xml. Internally, Dexter converts all the HTTP request parameters into an XML parameter document. The parameter document is encapsulated as a ParamBean (see 5.2.4) previously. During its execution, idoc_viewer.xml creates another idoc dynamically, by filling in an idoc template called dynamicinstr.xml (middle Figure 13), which it then executes to create the response document. The idoc dynamicinstr.xml simply inserts the requested idoc into a text area within an XHTML template document, idoctemplate.xml (bottom right).

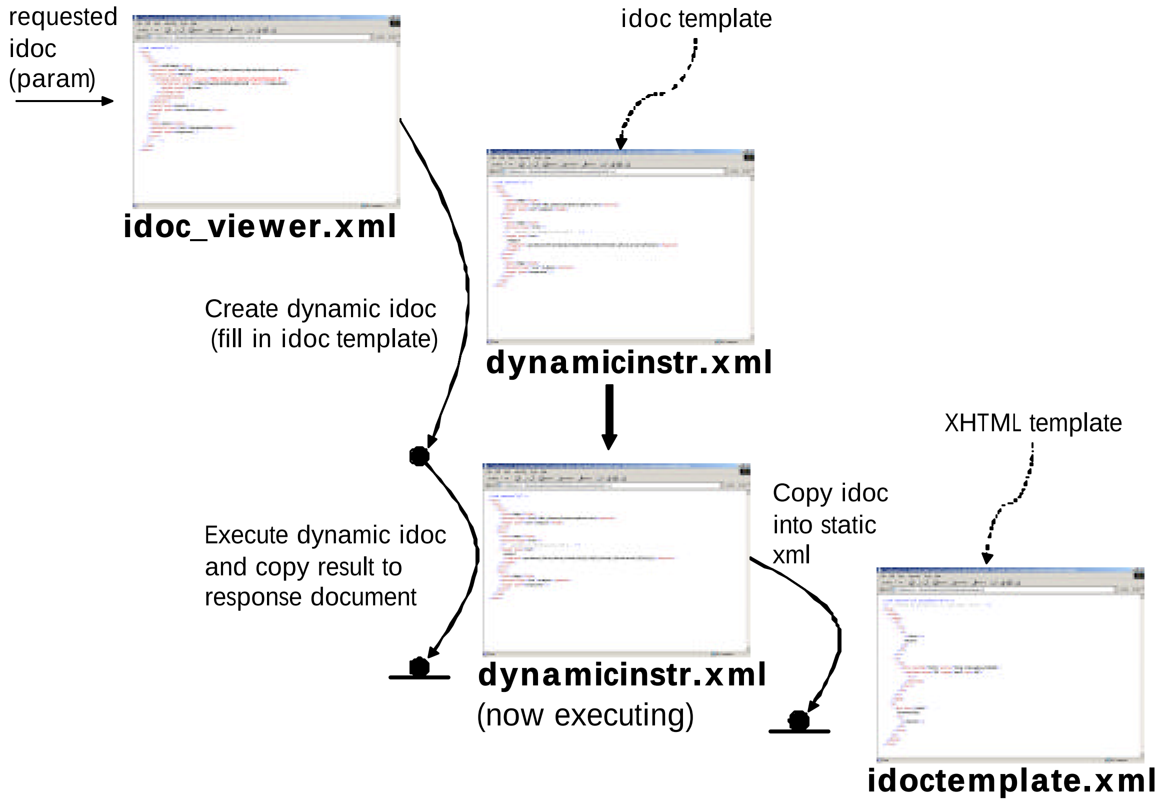
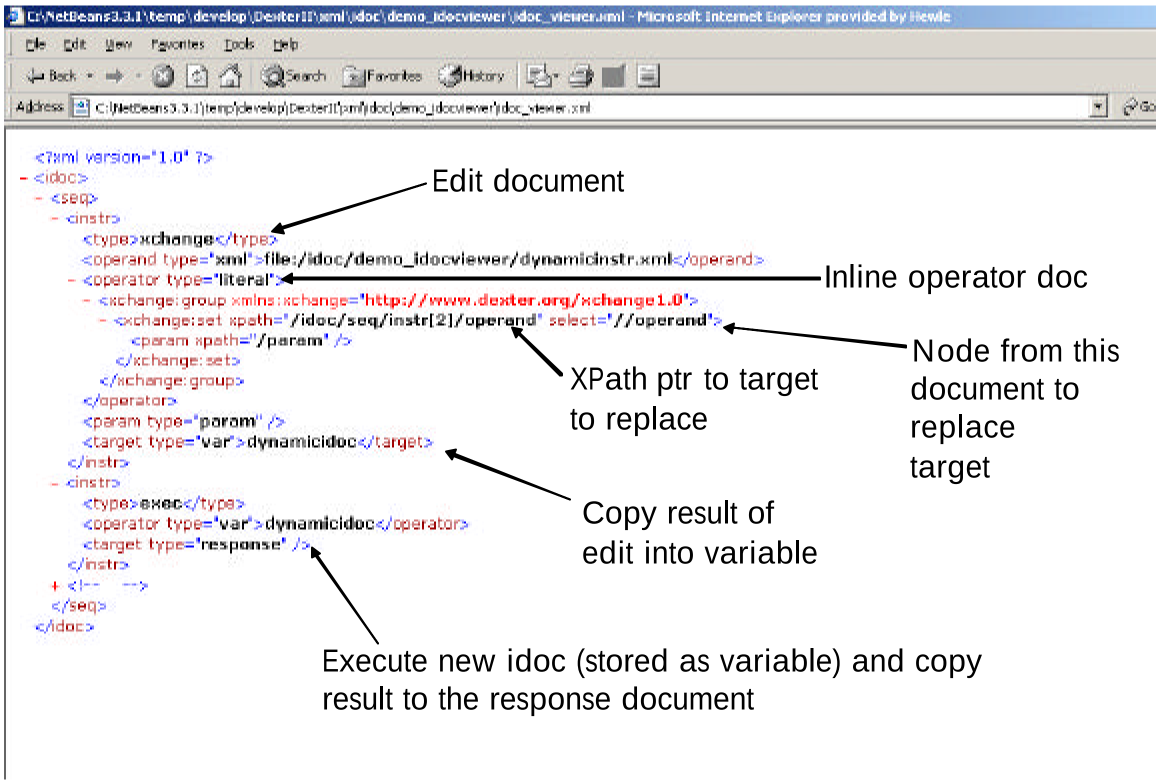

requested idoc (param)
idoc_viewer.xml
idoc template
dynamicinstr.xml
Create dynamic idoc (fill in idoc template)
XHTML template
Copy idoc into static xml
Execute dynamic idoc and copy result to response document
dynamicinstr.xml (now executing)
idoctemplate.xml

Figure 13: Overview of the response document creation process for viewing an idoc

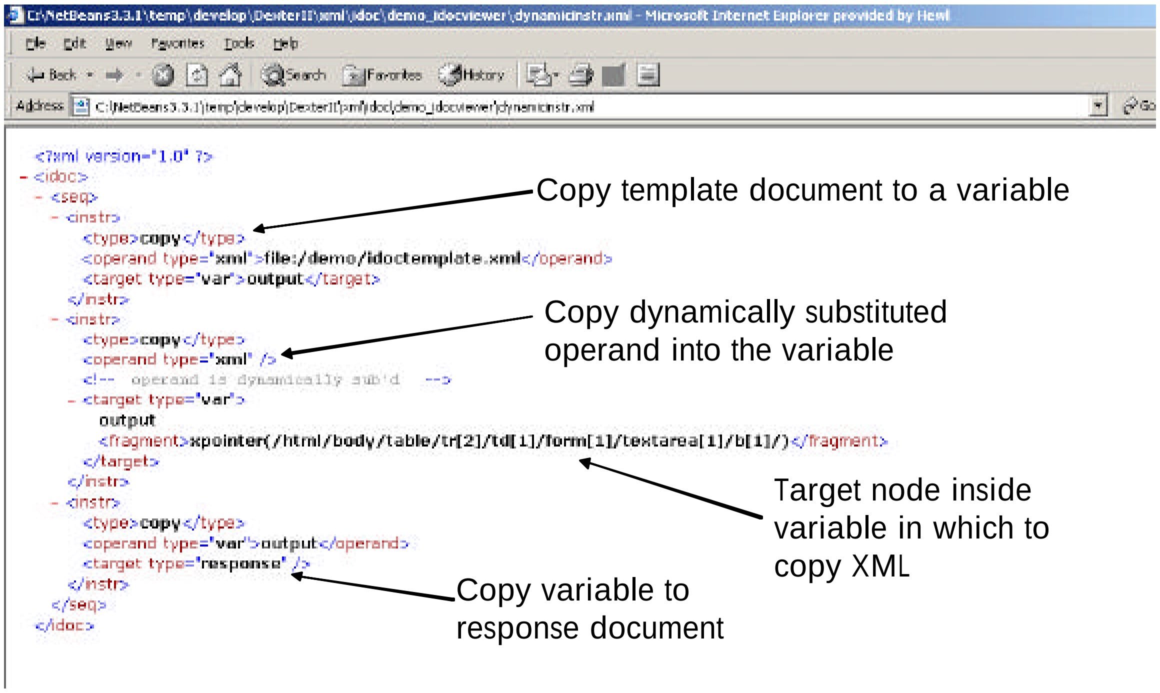
The three XML documents shown in Figure 13, are expanded in the following three figures. Idoc_viewer.xml, in Figure 14 contains two instructions. The first instruction is of type *xchange* and simply copies the idoc URI specified in the parameter bean into the dynamicinstr.xml idoc. The operator in the first instruction is of type *literal*, so the actual operator document is embedded as a fragment in the idoc. Internally Dexter creates a unique URI for the literal fragment and places it in the cache. The first instruction also contains a *parameter*, which in this case points to the parameter bean. Like the *response* reserved word, Dexter internally resolves this reference to the unique parameter bean URI associated with this idoc execution. Note that the parameter element inside an instruction can specify any URI-addressable XML document in the same way as the operand and operator.


C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_idocviewer\idoc_viewer.xml - Microsoft Internet Explorer provided by Hewle
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_idocviewer\idoc_viewer.xml
Go
Edit document
Inline operator doc
Node from this document to replace target
XPath ptr to target to replace
Copy result of edit into variable
Execute new idoc (stored as variable) and copy result to the response document

```
  <?xml version="1.0" ?>
- <idoc>
  - <seq>
    - <instr>
        <type>xchange</type>
        <operand type="xml">file:/idoc/demo_idocviewer/dynamicinstr.xml</operand>
      - <operator type="literal">
        - <xchange:group xmlns:xchange="http://www.dexter.org/xchange1.0">
          - <xchange:set xpath="/idoc/seq/instr[2]/operand" select="//operand">
              <param xpath="/param" />
            </xchange:set>
          </xchange:group>
        </operator>
        <param type="param" />
        <target type="var">dynamicidoc</target>
      </instr>
    - <instr>
        <type>exec</type>
        <operator type="var">dynamicidoc</operator>
        <target type="response" />
      </instr>
    + <!--   -->
    </seq>
  </idoc>
```

Figure 14: Example idoc which dynamically creates another idoc and executes it (idoc-viewer.xml)

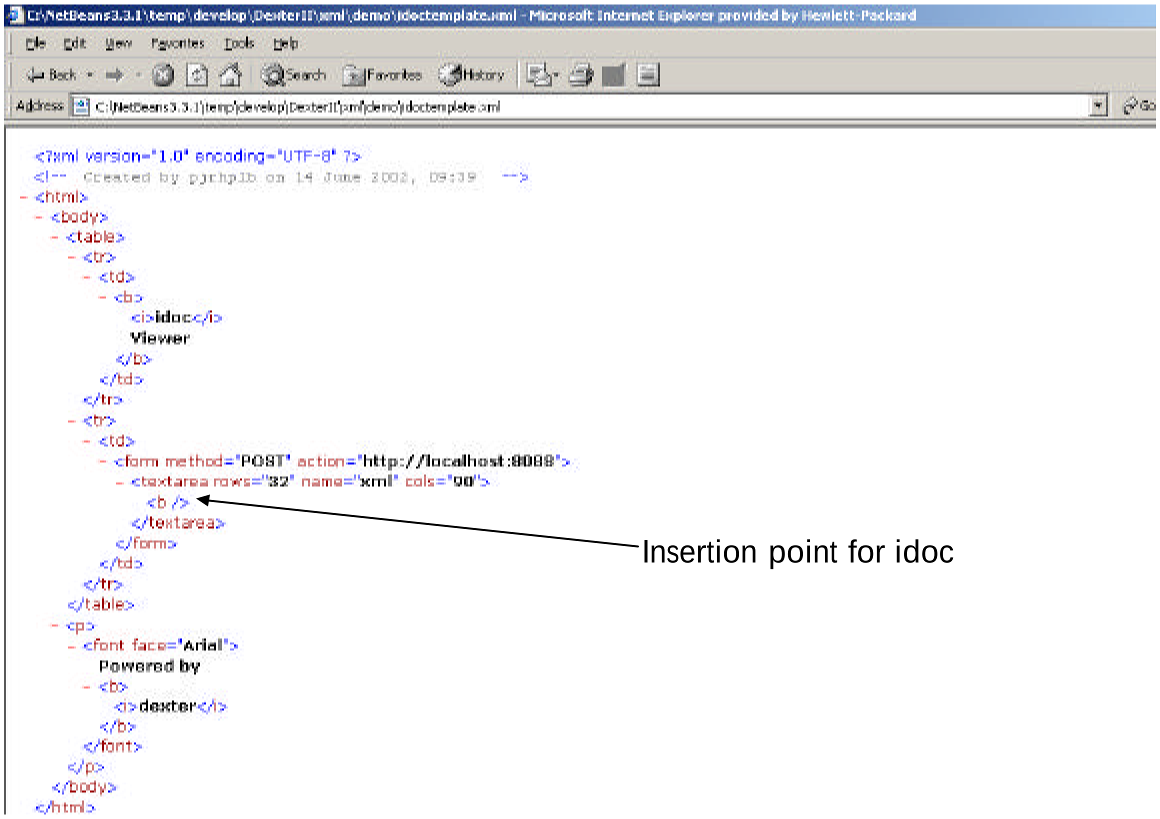
The dynamicinstr.xml *template* idoc is shown in Figure 15. This contains three instructions. The first simply creates a separate copy of the idoctemplate.xml document which is written to a variable named *output*. The second instruction then inserts the document specified by the operand into *output*. Note that the operand in the second instruction is intentionally blank and is filled in by the idoc_viewer.xml. In this example the document is inserted at a node within *output* specified by the XPath inside the *fragment* element in the target. Finally in the third instruction, the document now held by output is copied into the response document.


C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_idocviewer\dynamicinstr.xml - Microsoft Internet Explorer provided by Hewl
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_idocviewer\dynamicinstr.xml
Go
Copy template document to a variable
Copy dynamically substituted operand into the variable
Target node inside variable in which to copy XML
Copy variable to response document

```
  <?xml version="1.0" ?>
- <idoc>
  - <seq>
    - <instr>
        <type>copy</type>
        <operand type="xml">file:/demo/idoctemplate.xml</operand>
        <target type="var">output</target>
      </instr>
    - <instr>
        <type>copy</type>
        <operand type="xml" />
        <!--  operand is dynamically sub'd  -->
      - <target type="var">
          output
          <fragment>xpointer(/html/body/table/tr[2]/td[1]/form[1]/textarea[1]/b[1]/)</fragment>
        </target>
      </instr>
    - <instr>
        <type>copy</type>
        <operand type="var">output</operand>
        <target type="response" />
      </instr>
    </seq>
  </idoc>
```

Figure 15: dynamicinstr.xml, an idoc which fills in a static XHTML template

The actual XHTML template is very simple. As shown in Figure 16, it contains a text area where the idoc that the user has requested is inserted.

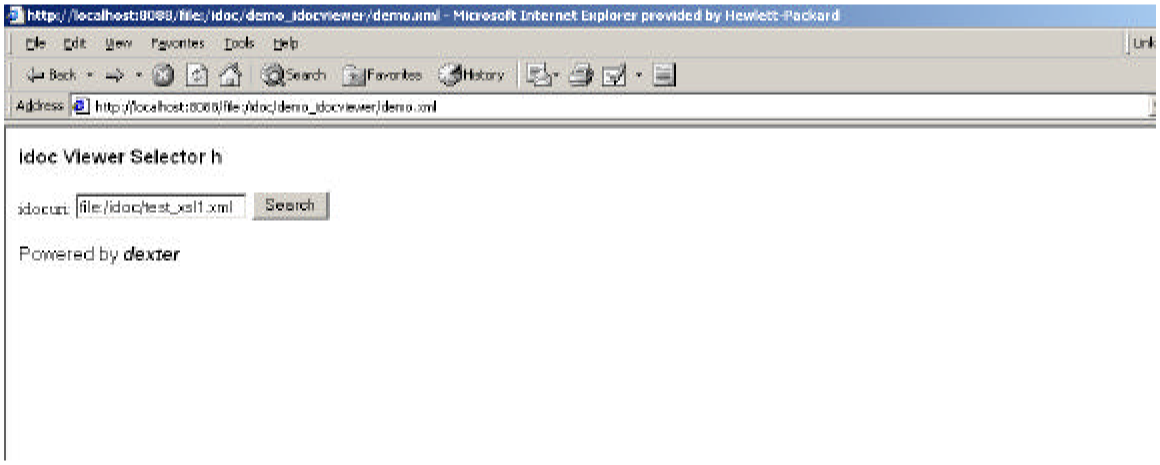

C:\NetBeans3.3.1\temp\develop\DexterII\xml\demo\idoctemplate.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\demo\idoctemplate.xml
Go
Insertion point for idoc

```
  <?xml version="1.0" encoding="UTF-8" ?>
  <!--  Created by pjchplb on 14 June 2002, 09:39  -->
- <html>
  - <body>
    - <table>
      - <tr>
        - <td>
          - <b>
              <i>idoc</i>
              Viewer
            </b>
          </td>
        </tr>
      - <tr>
        - <td>
          - <form method="POST" action="http://localhost:8088">
            - <textarea rows="32" name="xml" cols="90">
                <b />
              </textarea>
            </form>
          </td>
        </tr>
      </table>
    - <p>
      - <font face="Arial">
          Powered by
        - <b>
            <i>dexter</i>
          </b>
        </font>
      </p>
    </body>
  </html>
```

Figure 16: The static XHTML filled in by the idoc dynamicinstr.xml

6.3.3 The screen shot

Figure 17, shows the screen shot of the form to be filled in by the user. This screen shot relates to the idoc in Figure 12. In this case the user has specified that test_xsl.xml idoc is displayed.


http://localhost:8088/file:/idoc/demo_idocviewer/demo.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Address http://localhost:8088/file:/idoc/demo_idocviewer/demo.xml

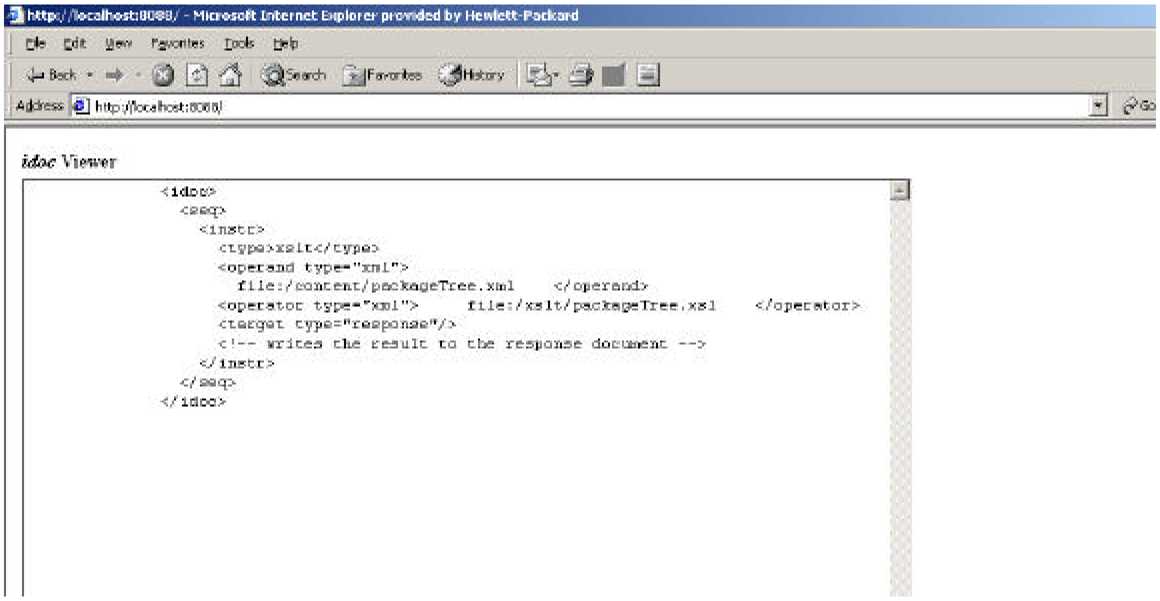
idoc Viewer Selector h

idocuri: file:/idoc/test_xsl1.xml Search

Powered by dexter

Figure 17: idoc viewer request form (filled in)

The output from executing the idoc, idoc_viewer.xml is shown below in Figure 18.

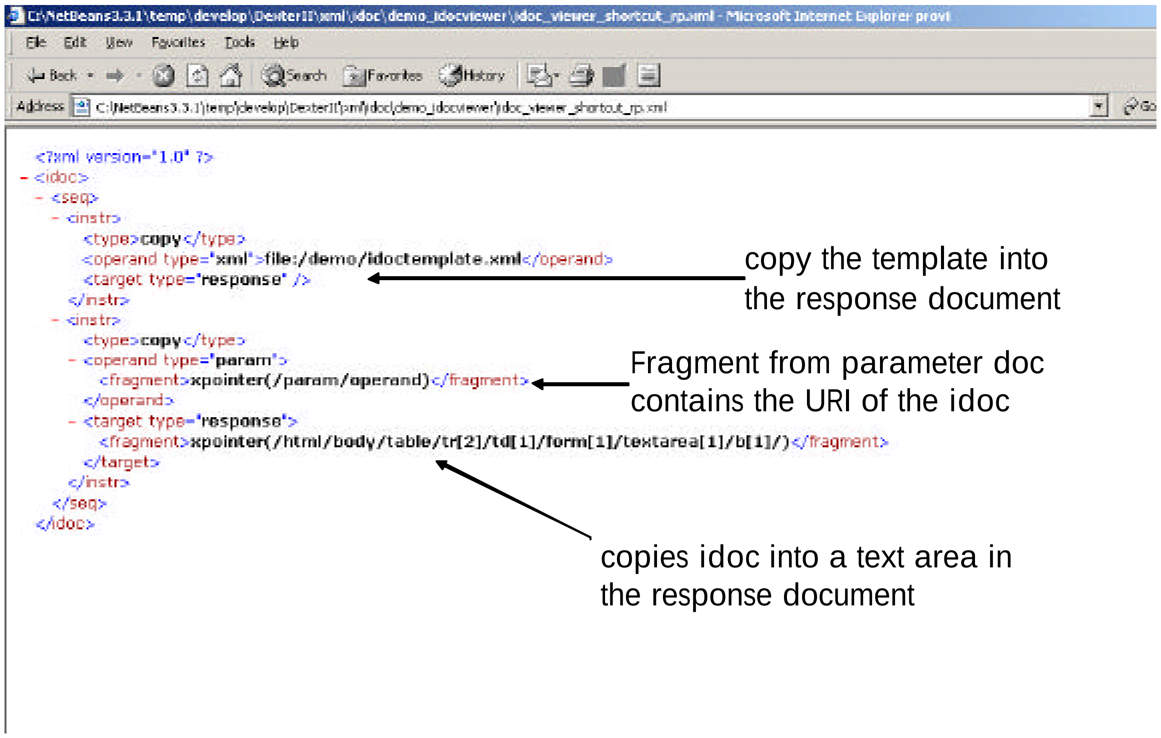

http://localhost:8088/ - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Address http://localhost:8088/

idoc Viewer

```
<idoc>
  <seq>
    <instr>
      <type>xslt</type>
      <operand type="xml">
        file:/content/packageTree.xml    </operand>
      <operator type="xml">      file:/xslt/packageTree.xsl    </operator>
      <target type="response"/>
      <!-- writes the result to the response document -->
    </instr>
  </seq>
</idoc>
```

Figure 18: Display of the requested idoc (file:/idoc/test_xsl1.xml)

As mentioned earlier, the idoc for generating Figure 18 could be simplified. In fact a single idoc, shown in Figure 19, can be used instead. It uses fragment identifiers in both the target and operand. The parameter document contains the reference to the idoc for display under the element named operand. The operand element name was assigned as part of the "natural" mapping from the HTTP request parameters to the parameter document; i.e. natural in the sense that the name of the XHTML form textbox was "*operand*". Since the root element of the parameter bean is named *param* by convention, the reference to the idoc is contained by the element given by the XPath expression "/param/operand".



```
C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_idocviewer\idoc_viewer_shortcut_rp.xml - Microsoft Internet Explorer provi
```

File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address
Go

```
C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demo_idocviewer\idoc_viewer_shortcut_rp.xml
```

```
  <?xml version="1.0" ?>
- <idoc>
  - <seq>
    - <instr>
        <type>copy</type>
        <operand type="xml">file:/demo/idoctemplate.xml</operand>
        <target type="response" />
      </instr>
    - <instr>
        <type>copy</type>
      - <operand type="param">
          <fragment>xpointer(/param/operand)</fragment>
        </operand>
      - <target type="response">
          <fragment>xpointer(/html/body/table/tr[2]/td[1]/form[1]/textarea[1]/b[1]/)</fragment>
        </target>
      </instr>
    </seq>
  </idoc>
```

copy the template into the response document
Fragment from parameter doc contains the URI of the idoc
copies idoc into a text area in the response document

Figure 19: Simpler idoc for displaying an idoc requested by a user through the web form shown in Figure 17

For the specific case of writing idocs built up from fragments of XHTML, the authoring process could be simplified by allowing the user to embed instructions directly inside an XHTML template page, from which an idoc and empty template document could easily be generated using idoc design tools.

6.4 Exception Handling

Exceptions and exception handling are important aids to building robust code and can be used to help debug code generally. This is no less true of debugging idocs. This 'example' is used to demonstrate how exceptions are caught and handled within Dexter.

6.4.1 Description

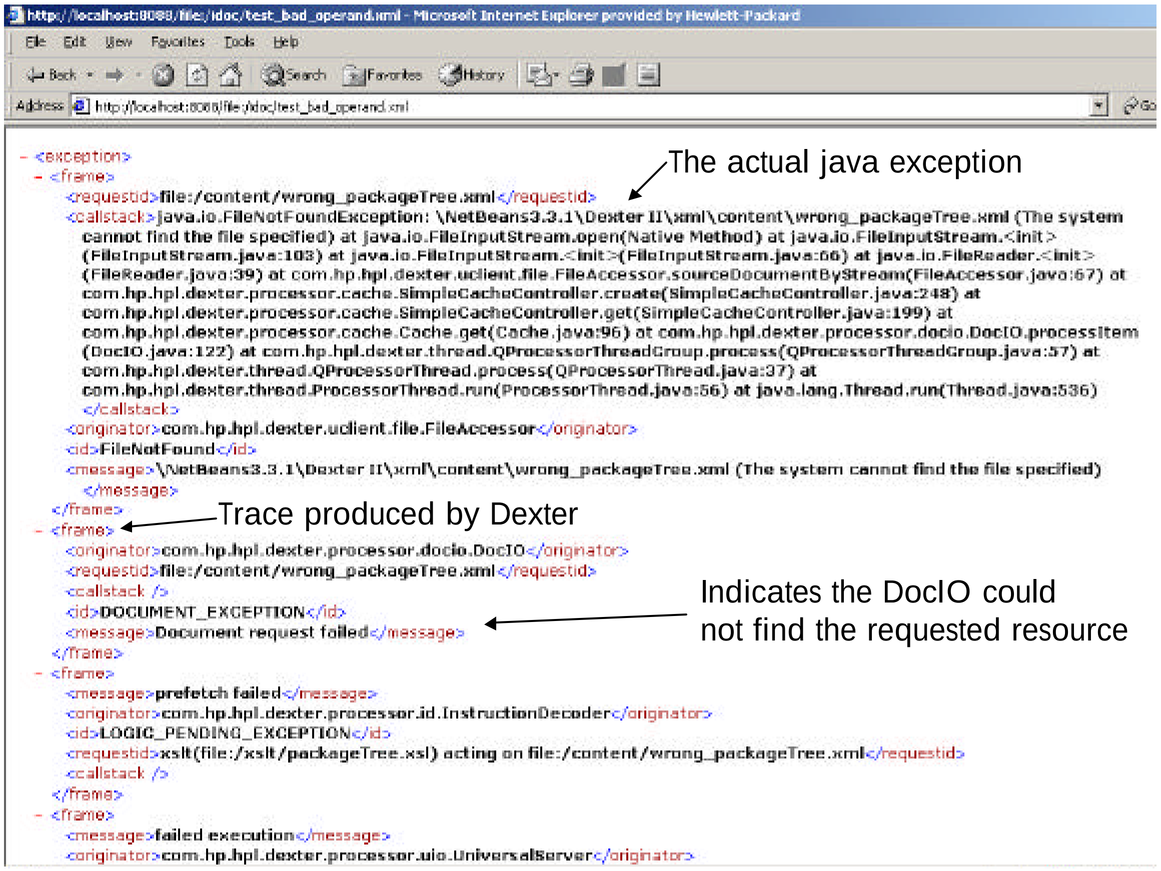
In this example, an idoc with a known error is executed; the error being an incorrectly specified operand. An exception document is produced in plain XML displaying the reason for failure. This is returned in the place of the response document. Thus, although Dexter allows exceptions to be caught, it currently does not provide support for writing exception handlers; exceptions are simply returned to the client.

6.4.2 The idoc

The idoc is not shown as it is intentionally invalid. It is, in fact, example 1 with the operand incorrectly specified. Instead of “packageTree2.xml”, the file “wrong_packageTree.xml” was specified that does not exist.

6.4.3 The screen shot

This screenshot is representative of the output from Dexter when an error occurs during execution of an idoc.



http://localhost:8088/file:/idoc/test_bad_operand.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8088/file:/idoc/test_bad_operand.xml
Go
The actual java exception
Trace produced by Dexter
Indicates the DocIO could not find the requested resource

```
- <exception>
  - <frame>
      <requestid>file:/content/wrong_packageTree.xml</requestid>
      <callstack>java.io.FileNotFoundException: \NetBeans3.3.1\Dexter II\xml\content\wrong_packageTree.xml (The system
        cannot find the file specified) at java.io.FileInputStream.open(Native Method) at java.io.FileInputStream.<init>
        (FileInputStream.java:103) at java.io.FileInputStream.<init>(FileInputStream.java:66) at java.io.FileReader.<init>
        (FileReader.java:39) at com.hp.hpl.dexter.uclient.file.FileAccessor.sourceDocumentByStream(FileAccessor.java:67) at
        com.hp.hpl.dexter.processor.cache.SimpleCacheController.create(SimpleCacheController.java:248) at
        com.hp.hpl.dexter.processor.cache.SimpleCacheController.get(SimpleCacheController.java:199) at
        com.hp.hpl.dexter.processor.cache.Cache.get(Cache.java:96) at com.hp.hpl.dexter.processor.docio.DocIO.processItem
        (DocIO.java:122) at com.hp.hpl.dexter.thread.QProcessorThreadGroup.process(QProcessorThreadGroup.java:57) at
        com.hp.hpl.dexter.thread.QProcessorThread.process(QProcessorThread.java:37) at
        com.hp.hpl.dexter.thread.ProcessorThread.run(ProcessorThread.java:56) at java.lang.Thread.run(Thread.java:536)
        </callstack>
      <originator>com.hp.hpl.dexter.uclient.file.FileAccessor</originator>
      <id>FileNotFound</id>
      <message>\NetBeans3.3.1\Dexter II\xml\content\wrong_packageTree.xml (The system cannot find the file specified)
        </message>
    </frame>
  - <frame>
      <originator>com.hp.hpl.dexter.processor.docio.DocIO</originator>
      <requestid>file:/content/wrong_packageTree.xml</requestid>
      <callstack />
      <id>DOCUMENT_EXCEPTION</id>
      <message>Document request failed</message>
    </frame>
  - <frame>
      <message>prefetch failed</message>
      <originator>com.hp.hpl.dexter.processor.id.InstructionDecoder</originator>
      <id>LOGIC_PENDING_EXCEPTION</id>
      <requestid>xslt(file:/xslt/packageTree.xsl) acting on file:/content/wrong_packageTree.xml</requestid>
      <callstack />
    </frame>
  - <frame>
      <message>failed execution</message>
      <originator>com.hp.hpl.dexter.processor.uio.UniversalServer</originator>
```

Figure 20: Output from Dexter when executing a invalid idoc

6.5 Displaying the Universal Server Statistics (Introspection 1)

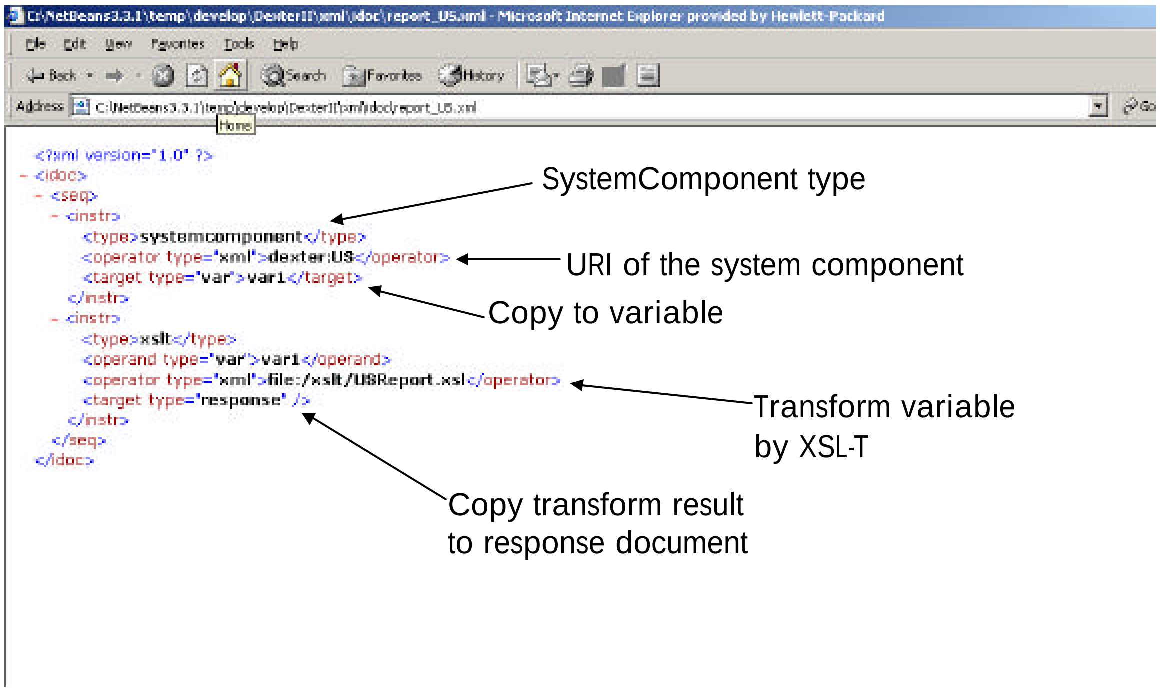
So far the example idocs have referenced XML resources stored as files on the local file system or as cached documents. This example illustrates how resources that are not natively stored in XML can be accessed and used within Dexter.

6.5.1 Description

This example illustrates how a display of the current status of the Universal Server component inside the Dexter XAP can be provided. To do this a new accessor type was created called the *systemcomponent*. This simply provides a mechanism for serializing the status information of a specified component to XML. All system components within Dexter are uniquely identified by a URI of the form dexter:<*component-name*>. In the case of the Universal Server, its URI is dexter:US.

6.5.2 The idoc

The usage statistics of the Universal Server are obtained in the first instruction and written to a variable *var1*, which is then transformed into XHTML and copied to the response document.

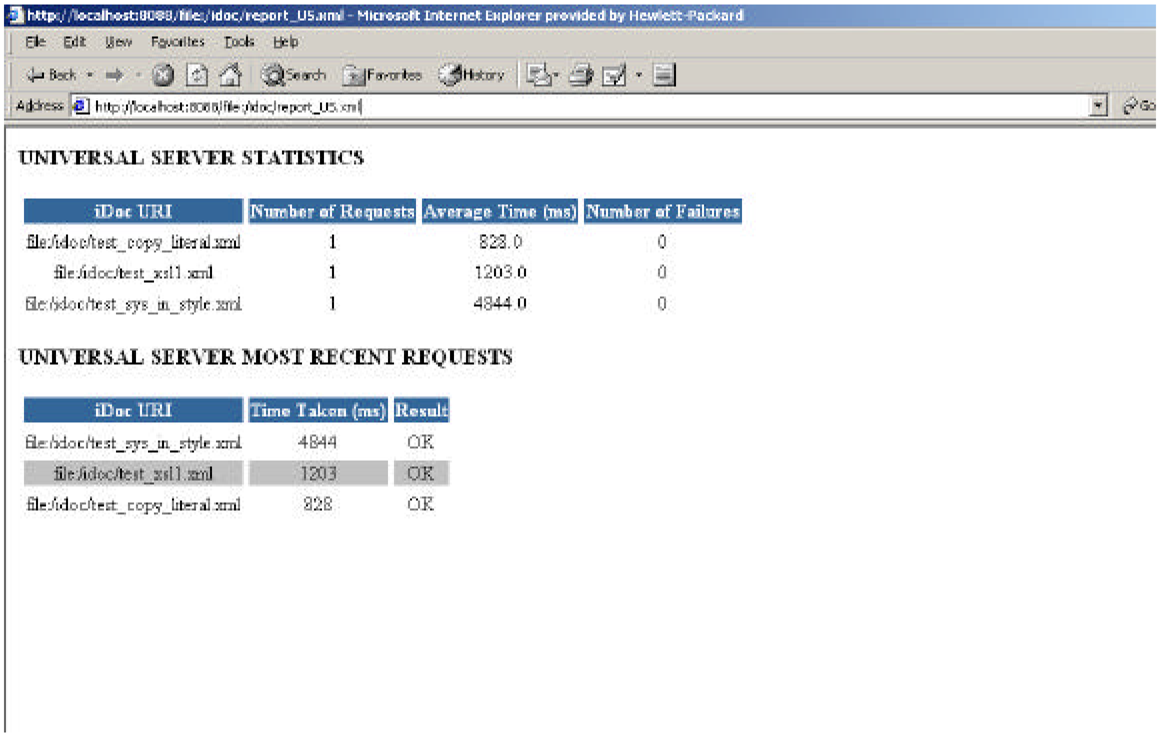

C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\report_US.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\report_US.xml
Home
Go
SystemComponent type
URI of the system component
Copy to variable
Transform variable by XSL-T
Copy transform result to response document

```
  <?xml version="1.0" ?>
- <idoc>
  - <seq>
    - <instr>
        <type>systemcomponent</type>
        <operator type="xml">dexter:US</operator>
        <target type="var">var1</target>
      </instr>
    - <instr>
        <type>xslt</type>
        <operand type="var">var1</operand>
        <operator type="xml">file:/xslt/USReport.xsl</operator>
        <target type="response" />
      </instr>
    </seq>
  </idoc>
```

Figure 21: idoc for displaying the Universal Server Statistics in XHTML

6.5.3 The screen shot

The output is shown in Figure 22. Two sets of statistics are gathered by the Universal Server and are presented as two separate tables. The top table contains a list of all idocs that have been executed since booting, the numbers of times a given idoc has been executed, and the average time taken to execute each idoc. The top table also indicates the number of idoc execution failures that have occurred. The bottom table simply lists the last three idocs that have been executed, the time taken to execute them and whether the execution completed successfully. Since each idoc in the bottom table has only been executed once, the average time and the last time taken are identical.


http://localhost:8088/file:/idoc/report_US.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8088/file:/idoc/report_US.xml
Go

UNIVERSAL SERVER STATISTICS

iDoc URI	Number of Requests	Average Time (ms)	Number of Failures
file:/idoc/test_copy_literal.xml	1	828.0	0
file:/idoc/test_xsl1.xml	1	1203.0	0
file:/idoc/test_sys_in_style.xml	1	4844.0	0

UNIVERSAL SERVER MOST RECENT REQUESTS

iDoc URI	Time Taken (ms)	Result
file:/idoc/test_sys_in_style.xml	4844	OK
file:/idoc/test_xsl1.xml	1203	OK
file:/idoc/test_copy_literal.xml	828	OK

Figure 22: Most recent Universal Server Statistics displayed in XHTML

6.6 System Introspection 2

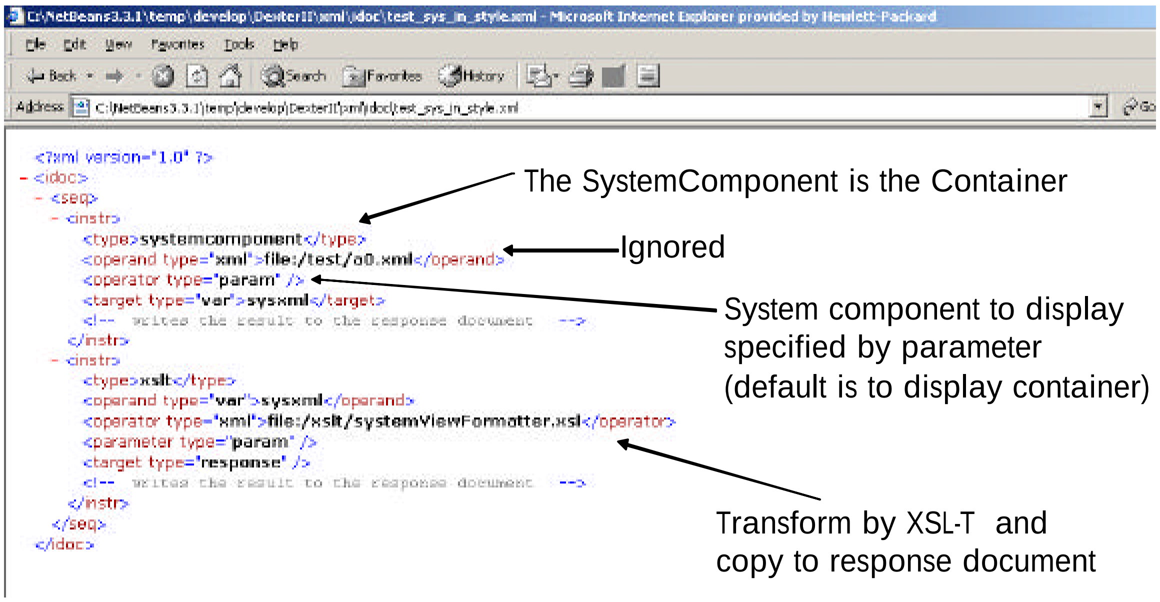
This is another example of system introspection which, but which also includes user input.

6.6.1 Description

In this example a system view is provided which contains details of the cache status and the number of threads currently used inside the container.

6.6.2 The idoc

The structure of this idoc is very similar to the previous idoc. As before, the instruction type is specified to be *systemcomponent*, but the component to view is specified in the parameter bean. In this case, the whole parameter document is passed to the accessor rather than just a fragment from within the parameter bean as in the example in section 6.3. The serialized system view formatted in XML is stored in a temporary variable called *sysxml* which is then transformed to produce the XHTML view.

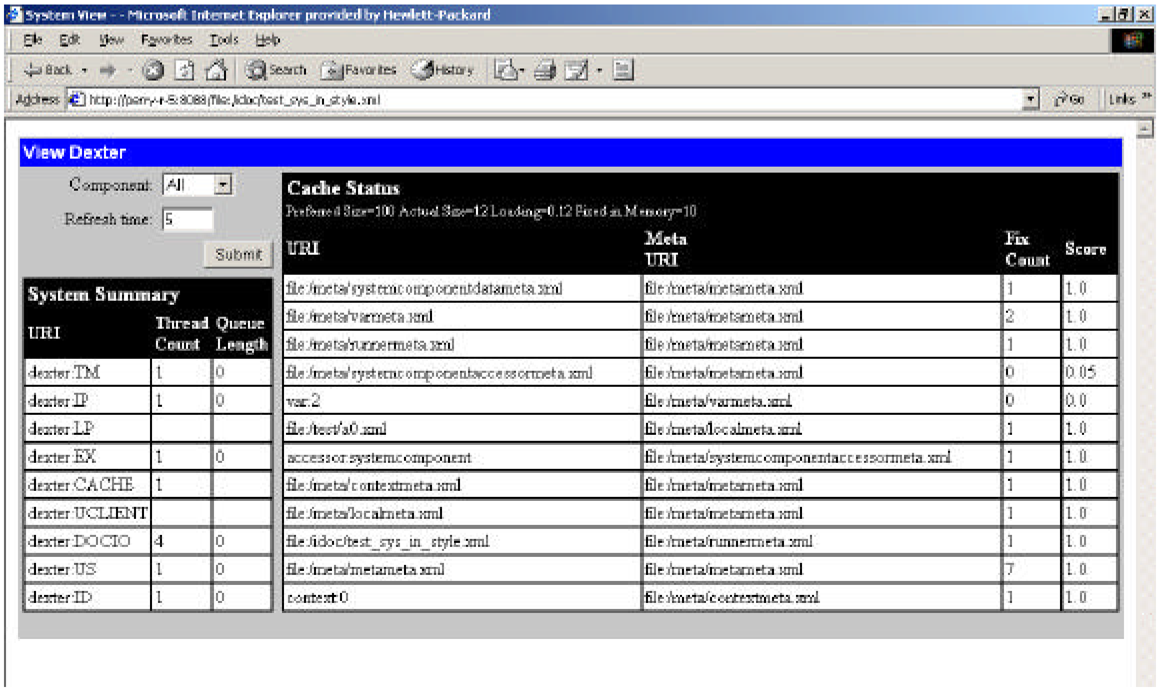

C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\test_sys_in_style.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\test_sys_in_style.xml
Go
The SystemComponent is the Container
Ignored
System component to display specified by parameter (default is to display container)
Transform by XSL-T and copy to response document

```
  <?xml version="1.0" ?>
- <idoc>
  - <seq>
    - <instr>
        <type>systemcomponent</type>
        <operand type="xml">file:/test/a0.xml</operand>
        <operator type="param" />
        <target type="var">sysxml</target>
        <!--  writes the result to the response document   -->
      </instr>
    - <instr>
        <type>xslt</type>
        <operand type="var">sysxml</operand>
        <operator type="xml">file:/xslt/systemViewFormatter.xsl</operator>
        <parameter type="param" />
        <target type="response" />
        <!--  writes the result to the response document   -->
      </instr>
    </seq>
  </idoc>
```

Figure 23: idoc to display all the container components

6.6.3 The screen shot

This shot shows the status of the cache and the number of running threads inside the XAP. The System Summary table shows the number of threads assigned to each component and the number of *jobs* pending in each component queue (if applicable). The Cache status shows the URIs of cached XMLbeans, the number of references to those beans (fix count) and a score (0..1), which indicates whether the bean is a candidate for removal from the cache; a score of 0 means that the bean can be removed safely, a score of 1 means that the bean may not be removed. In addition the cache table includes the URI to the Metabean of each of the XMLBeans. The MetaMetaBean is its own MetaBean (as described in 5.2.5) and has the greatest number of references, as would be expected.


System View - - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://perry-r-5:8088/file:/idoc/test_sys_in_style.xml
Go Links
View Dexter

Component: All

Refresh time: 5

Submit

System Summary

URI	Thread Count	Queue Length
dexter:TM	1	0
dexter:IP	1	0
dexter:LP		
dexter:EX	1	0
dexter:CACHE	1	
dexter:UCLIENT		
dexter:DOCIO	4	0
dexter:US	1	0
dexter:ID	1	0

Cache Status

Preferred Size=100 Actual Size=12 Loading=0.12 Fixed in Memory=10

URI	Meta URI	Fix Count	Score
file:/meta/systemcomponentdatameta.xml	file:/meta/metameta.xml	1	1.0
file:/meta/varmeta.xml	file:/meta/metameta.xml	2	1.0
file:/meta/runnermeta.xml	file:/meta/metameta.xml	1	1.0
file:/meta/systemcomponentaccessormeta.xml	file:/meta/metameta.xml	0	0.05
var:2	file:/meta/varmeta.xml	0	0.0
file:/test/a0.xml	file:/meta/localmeta.xml	1	1.0
accessor:systemcomponent	file:/meta/systemcomponentaccessormeta.xml	1	1.0
file:/meta/contextmeta.xml	file:/meta/metameta.xml	1	1.0
file:/meta/localmeta.xml	file:/meta/metameta.xml	1	1.0
file:/idoc/test_sys_in_style.xml	file:/meta/runnermeta.xml	1	1.0
file:/meta/metameta.xml	file:/meta/metameta.xml	7	1.0
context:0	file:/meta/contextmeta.xml	1	1.0

Figure 24: View of all the container components

6.7 Showing use of XQuery

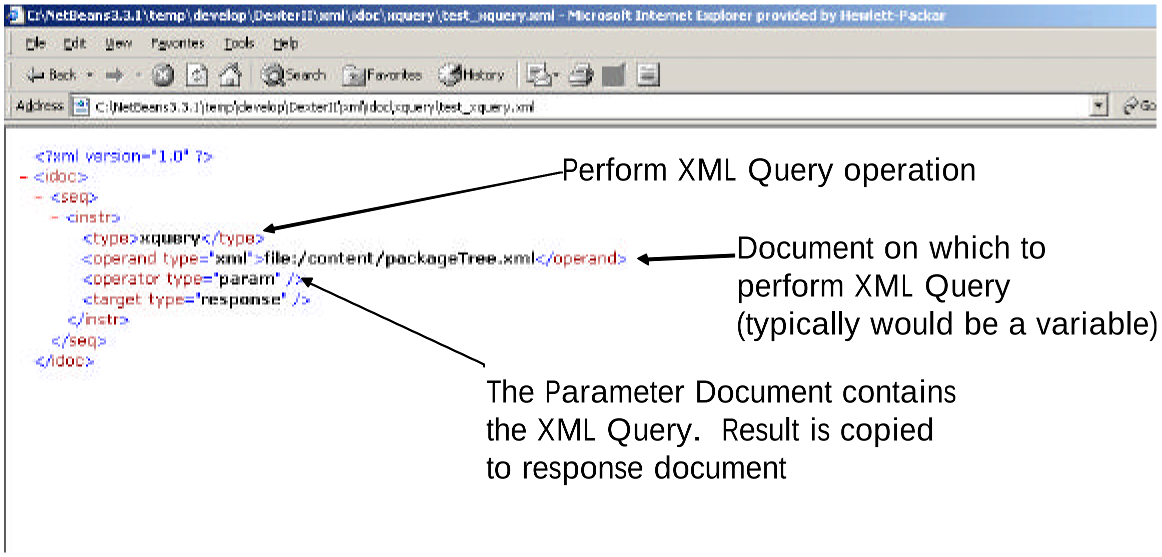
This example demonstrates integration of another type of XML standard-based accessor other than XSL-T.

6.7.1 Description

Dexter's XQuery active accessor is implemented using the XCool XQuery engine [29]. In this example, an XML Query is performed over the Dexter source tree hierarchy. The query is submitted by a form.

6.7.2 The idoc

In this example, only the idoc required to process the XML Query is shown. All queries are performed on a single file "packageTree.xml", and the query submitted in the web form is passed through to the idoc using the ParameterBean.


C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\xquery\test_xquery.xml - Microsoft Internet Explorer provided by Hewlett-Packar
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\xquery\test_xquery.xml
Go

```
<?xml version="1.0" ?>
- <idoc>
  - <seq>
    - <instr>
        <type>xquery</type>
        <operand type="xml">file:/content/packageTree.xml</operand>
        <operator type="param" />
        <target type="response" />
      </instr>
    </seq>
  </idoc>
```

Perform XML Query operation

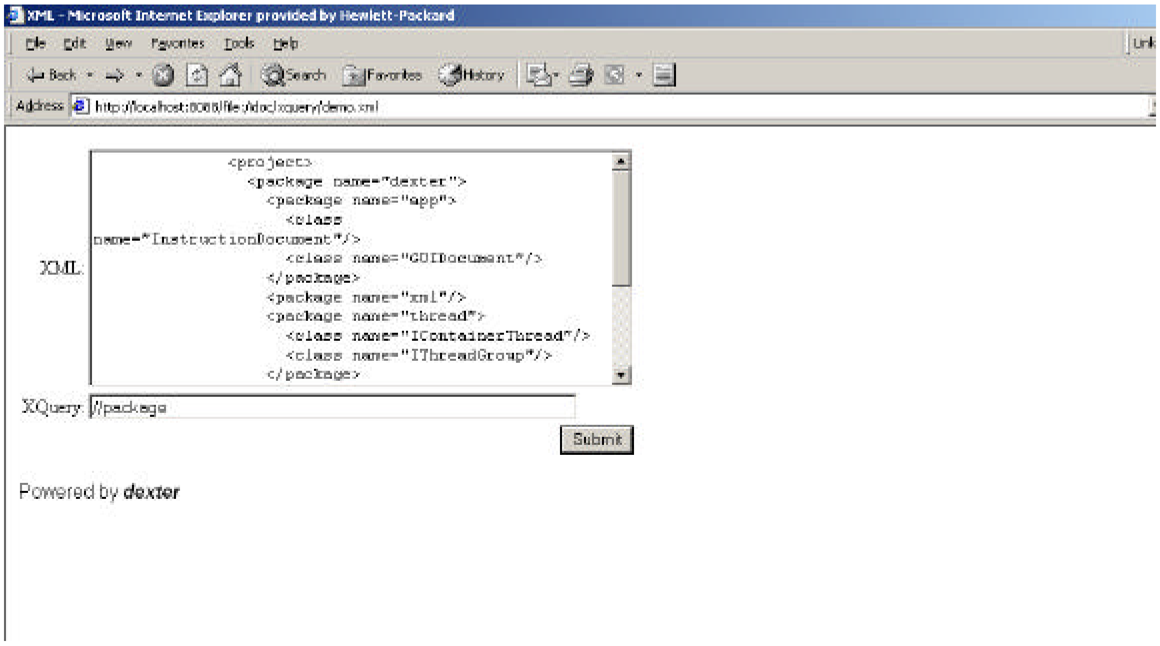
Document on which to perform XML Query (typically would be a variable)

The Parameter Document contains the XML Query. Result is copied to response document

Figure 25: idoc to process an XML Query over the package tree hierarchy.

6.7.3 The screen shot

Figure 26 shows the form through which the XQuery submission is made. Figure 27 displays the XQuery result as XML.


XML - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Links
Back Search Favorites History
Address http://localhost:8088/file:/idoc/xquery/demo.xml

XML:

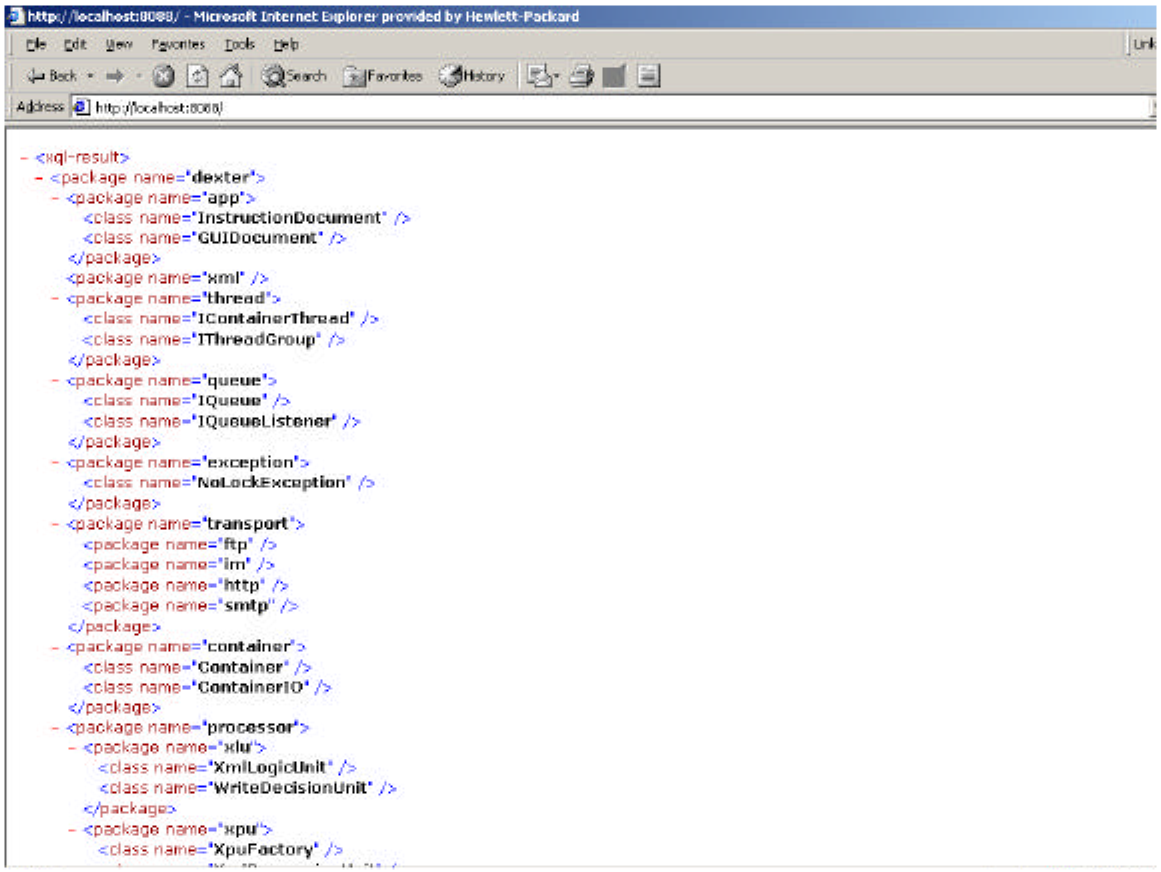
```
              <project>
                <package name="dexter">
                  <package name="app">
                    <class
name="InstructionDocument"/>
                    <class name="GUIDocument"/>
                  </package>
                  <package name="xml"/>
                  <package name="thread">
                    <class name="IContainerThread"/>
                    <class name="IThreadGroup"/>
                  </package>
```

XQuery: //package

Submit

Powered by *dexter*

Figure 26: XQuery Submission Interface


http://localhost:8088/ - Microsoft Internet Explorer provided by Hewlett-Packard
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8088/
Links

```
- <xql-result>
  - <package name="dexter">
    - <package name="app">
        <class name="InstructionDocument" />
        <class name="GUIDocument" />
      </package>
      <package name="xml" />
    - <package name="thread">
        <class name="IContainerThread" />
        <class name="IThreadGroup" />
      </package>
    - <package name="queue">
        <class name="IQueue" />
        <class name="IQueueListener" />
      </package>
    - <package name="exception">
        <class name="NoLockException" />
      </package>
    - <package name="transport">
        <package name="ftp" />
        <package name="im" />
        <package name="http" />
        <package name="smtp" />
      </package>
    - <package name="container">
        <class name="Container" />
        <class name="ContainerIO" />
      </package>
    - <package name="processor">
      - <package name="xlu">
          <class name="XmlLogicUnit" />
          <class name="WriteDecisionUnit" />
        </package>
      - <package name="xpu">
          <class name="XpuFactory" />
```

Figure 27: Result of the XQuery from Figure 26

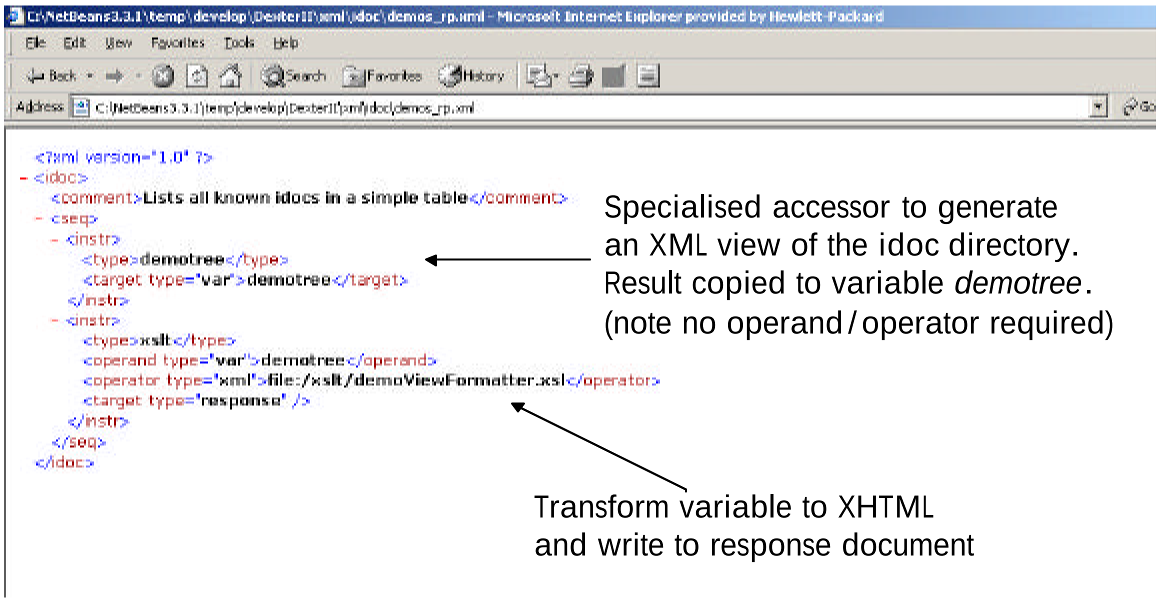
6.8 Viewing all the idoc demos

6.8.1 Description

This last example is used to display all the idoc examples written for system testing and demonstration.

6.8.2 The idoc

The idoc is shown in Figure 28. It is very similar to that used for displaying the Dexter source tree, but the accessor in this case dynamically creates the XML from the *idoc source directory* as part of every idoc execution. A specialised accessor called the *demotree* accessor has been created to walk the idoc source tree. Of course, a more generalised tree walker Accessor could easily be written to accept an arbitrary directory start point and various options such as file filters.



```
<?xml version="1.0" ?>
- <idoc>
    <comment>Lists all known idocs in a simple table</comment>
  - <seq>
    - <instr>
        <type>demotree</type>
        <target type="var">demotree</target>
      </instr>
    - <instr>
        <type>xslt</type>
        <operand type="var">demotree</operand>
        <operator type="xml">file:/xslt/demoViewFormatter.xsl</operator>
        <target type="response" />
      </instr>
    </seq>
  </idoc>
```

C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demos_rp.xml - Microsoft Internet Explorer provided by Hewlett-Packard
Address C:\NetBeans3.3.1\temp\develop\DexterII\xml\idoc\demos_rp.xml

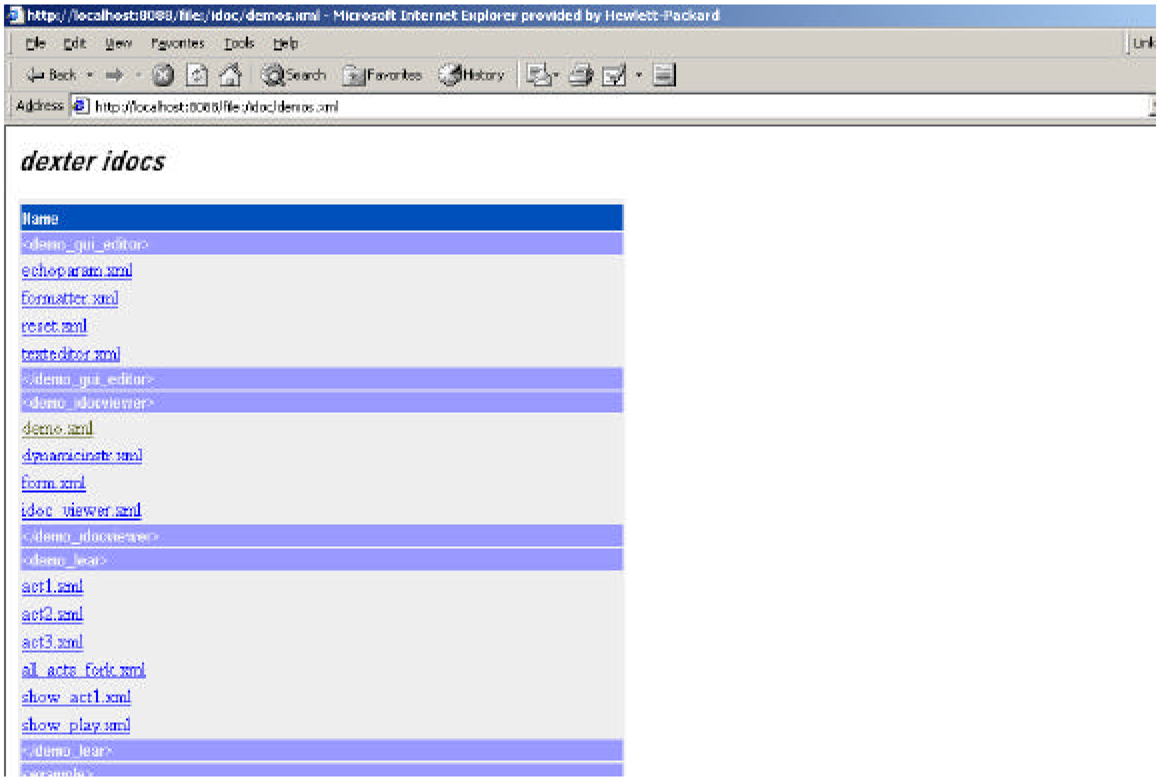
Specialised accessor to generate an XML view of the idoc directory. Result copied to variable *demotree*. (note no operand / operator required)

Transform variable to XHTML and write to response document

Figure 28: idoc workflow to display all the idocs in the Dexter root directory

6.8.3 The screen shot

The idocs found in the source tree are listed below. The XSL-T transformation also creates a URL link around each idoc name so that the user can easily click to execute any of the listed idocs. This helps facilitate testing.



http://localhost:8088/file:/idoc/demos.xml - Microsoft Internet Explorer provided by Hewlett-Packard
Address http://localhost:8088/file:/idoc/demos.xml

dexter idocs

Name
<demo_gui_editor>
echoparam.xml
formatter.xml
reset.xml
texteditor.xml
</demo_gui_editor>
<demo_idocviewer>
demo.xml
dynamicinstr.xml
form.xml
idoc_viewer.xml
</demo_idocviewer>
<demo_lear>
act1.xml
act2.xml
act3.xml
all_acts_fork.xml
show_act1.xml
show_play.xml
</demo_lear>

Figure 29: An XHTML display of the idocs contained in the container root directory

6.9 Conditional Execution

The previous idoc examples in this section do not include the use of conditional loops. Whilst they are supported within the idoc language schema, they have not been fully implemented at the time of writing. However to illustrate how they can be used, an example is shown below of a while loop. A new accessor type is required called the *Eval* accessor which evaulates an XPath expression against the operand. In the example below the XPath expression to evaluate, suitably escaped, is included in the operator element as a literal.

```
<idoc>
        <seq>
                <while>
                        <cond>
                                <instr>
                                        <type>eval</type>
                                        <operand type=”var”>var1</operand>
                                        <operator type=”literal”>
                                                <eval>
                                                        root/number &gt; 100
                                                </eval>
                                        </operator>
                                        <target type=”var”>var2</target>
                                <instr>
                        </cond>
                        <seq>
                                while true do instructions here
                        </seq>
                </while>
        <seq>
</idoc>
```

In this example, a test is made to see if the value of the element identified by the XPath root/number in document ***var1*** is greater then 100. The result either true or false, is written to a *BooleanBean* with the variable name ***var2***. If true, the instructions contained in the sequence element inside the while element are executed.

In order that the loop terminates, the idoc instructions within the while loop will need to change the number encapsulated within the var1 document to be less than 100.

It is stressed that idocs use conditional expressions sparingly as they can increase the complexity of the idoc. If a complex piece of functionality needs to be implemented then it is recommended to encapsulate it within an Active Accessor. However, for simple cases it is useful to provide conditional processing.

7 Futures

It is hoped that the examples from the previous section will demonstrate the scope and flexibility of Dexter and the idoc language. Before finishing however, we wish to include some illustrations of where Dexter could find application using slightly richer examples. In the first example we consider a hotel booking service which illustrates the how results of querying web services might be aggregated.

The operations performed in the Dexter container are shown flowing left to right in Figure 30. The sequence of operations can be easily expressed as an idoc. Initially a user request for room availability is made to two hypothetical services defined say using WSDL [7]. Using the WSDL definitions requests are made to two fictional hotel chains (Holiday House and Euro Hols) to check for room availability. The response (XML documents) are then transformed into a suitable common syntax (for querying) and added to a shared temporary document (*tmp doc*). This temporary document is then queried using the parameters supplied by the user. Finally the query results are transformed using an XSL-T transformation to produce the XHTML response document, which is returned to the user.

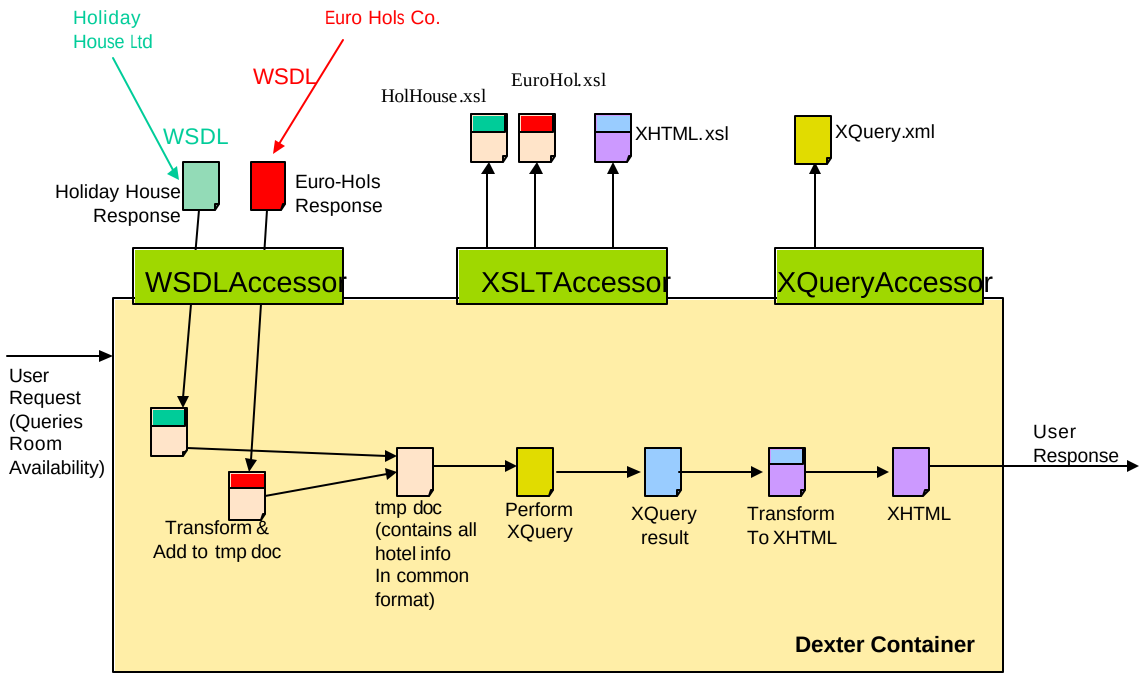


Figure 30: Using Dexter to aggregate web services and display a web page

The flow of operations shown in Figure 30 can all be supported using Dexter. Figure 30 shows the required accessor classes and the documents required to perform the sequence of operations as illustrated. Note that the two requests to the holiday booking services can be made concurrently by setting the instruction attribute *asynch* to true. Furthermore the caching policies applied to the response documents can provide for them to age them over say several minutes thereby avoiding the high latency external service invocations for every query. This of course assumes the hotel data changes relatively slowly. As a result, the transformations of the WSDL service responses are also cached, which makes subsequent queries much more rapid.

Our second example of a future use for Dexter is in the area of distributed graphical web applications. So far we have discussed XML applications that either do without a conventional graphical UI, or which use HTML in browsers as the UI medium. There are several initiatives underway at present that attempt to standardise *declarative* methods of describing graphical user interfaces and the events they must propagate and handle to realise an application. UIML [25] is one of these. XForms [18] and Web Services for Interactive Applications [17] share similar objectives. We conceive of using Dexter as the system which hosts and operates a distributed application whose GUI is described in, say UIML.

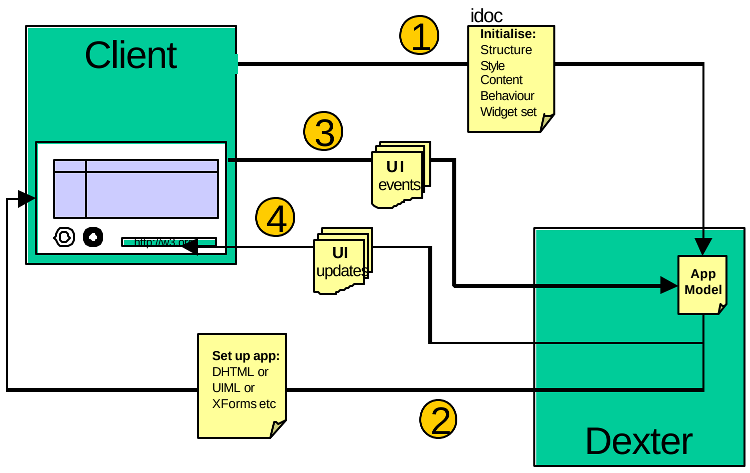


Figure 31: Dexter and a UIML driven Client application

Figure 31 above shows Dexter driving a distributed application with a GUI that is rendered by a special purpose client. The application starts by sending an idoc that invokes an initialisation stage (Step 1). The idoc references an *application model* document, which contains a canonical representation of the UI suitable for use with all envisaged client devices. Parameters in the incoming idoc may used to key the widget set and modality appropriate for the requesting client. The resulting markup is returned to the client (Step 2). User actions at the client cause *events* and *parameters* to be sent back to Dexter (Step 3) where they are correlated with static and dynamic updates that are again returned to the client (Step 4).

The final example shown here is the illustration of how an idoc could be used to define a workflow for generating a pdf document from an XML source. This is an interesting application since it produces non-XML output by using XSL Formatting Objects technology [7]. The example shown in Figure 34, shows a simple bill conversion from XML into PDF.

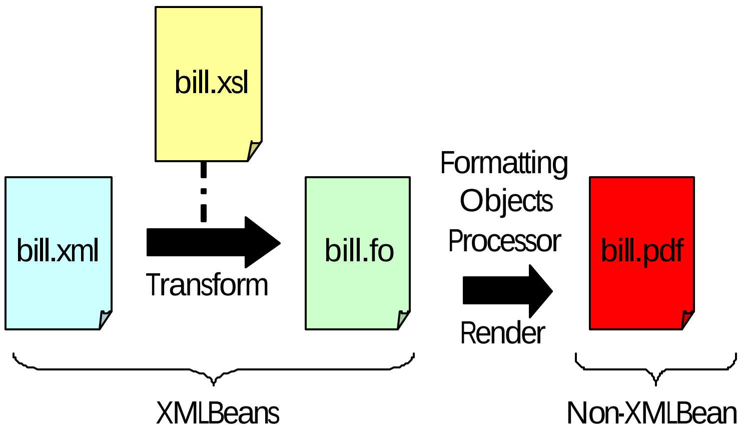


Figure 32: Generating a PDF document using Dexter

The first stage of the pipeline takes the raw subscriber billing data (bill.xml), expressed in XML, and applies a transform (bill.xsl) to convert it into a Formatting Objects XML document (bill.fo). Next, a formatting objects processor (FOP) converts the "bill.fo" document into a pdf document bill.pdf. To implement this functionality a FOPAccessor would be required, as well as a *PDFBean* to encapsulate the PDF output. It would be straightforward to adapt the system to handle non-XML beans to support this kind of functionality. The bill.pdf document could be returned as the response document, attaching it to an HTTP or SMTP response.

It is hoped that the three examples illustrated in this section will serve as motivators for other possible XML-rich applications. Whilst many of the examples contained in this document suggest a server-side bias for the XAP, this should not be seen as the case. The XAP can be deployed as a client side auxiliary component providing support XML processing. In fact without an HTTP transport the XAP can be *dropped in* a servlet, which then passes requests through to the XAP for processing.

8 Conclusion

We have described the architecture of an *XML Application Platform,* and Dexter – a research prototype XAP we have built. The motivation for the work was the observation that global (or Internet-wide) interoperability is an issue for many trends in IT: Web Services, B2B interactions, large scale Enterprise Application Integration and so on. XML is consolidating into a practical solution for data interoperability but less so for the description of data processing. Meanwhile, legacy 3-tier systems (the ubiquitous Web server-Middleware-Relational Database systems that underpin much current Web technology) are adapting to data portability in isolated proprietary ways. Task specific workflow languages are also proliferating [28] while standardisation efforts [1, 14, 26, 27] are moving painfully slowly towards a convergence.

We note that XML processing is now a de facto capability of all popular mainframe and PC Operating Systems. The Dexter project is a research probe into the feasibility and the utility of an XML Application Platform that allows much of the computation that is currently done in proprietary or language and platform specific ways, to be performed *in the XML domain*. We have found that using the XAP requires some re-orientation on the part of the developer to adjust to this new computational model. However it encourages reuse of XML technologies and supports building of complex applications in a modular way.

It is worth emphasising again that Dexter is *not* trying to replace conventional programming languages and isn't a procedural or object oriented language marked up in XML. All operands are treated as XML documents and therefore all operations are performed in the XML domain. This is a significantly different approach to application development and is attractive for web service development and other applications which involve document exchange and manipulation using languages like XSL-T or XQuery.

An interesting aspect of the popularity of XML is that it is a measure of the value the IT community places on open and transferable data. XML is in some ways less efficient than proprietary binary formats, and yet it has rapidly become the medium of choice for externalisation of data. There has always been a tension for business organisations between exposing the data they must to support transactions, and guarding their proprietary secrets. The adoption of XML illustrates the push toward the former. Dexter is aimed at making it also possible to externalise globally interoperable data *processing*.

We have refined a set of design principles (described in section 3) for XML Application Platforms, which were certainly helpful when we were building our prototype. It is our hope that we and others will be able to build on these principles and encourage the growth of XML Applications.

9 Acknowledgements

This work has benefited from numerous conversations with colleagues at HP Labs Bristol.

The authors would particularly like to acknowledge Tony Butterfield, a contractor who worked on the design and coding of the prototype. Tony brought a fresh perspective and made invaluable contributions.

The authors would also like to thank Dave Banks for a very careful reading of an earlier draft of this document which resulted in many improvements.

10 Appendix A: Dexter Source Tree

This appendix contains the complete package hierarchy of Dexter. Because of the large number of classes involved, a set of screen shots, all generated using Dexter, are used to display sub-parts of the package hierarchy a bit at a time. The top-level package, namely "com.hp.hpl.dexter" has been omitted to simplify the display, as have the test packages.

The console package contains classes related to the console application shown in Figure 7. The container package contains the top-level XAP platform piece. All resources are managed from within the container. The nbio package contains a separate transport module built upon the Java JDK 1.4 support non-blocking I/O API. This was never integrated.

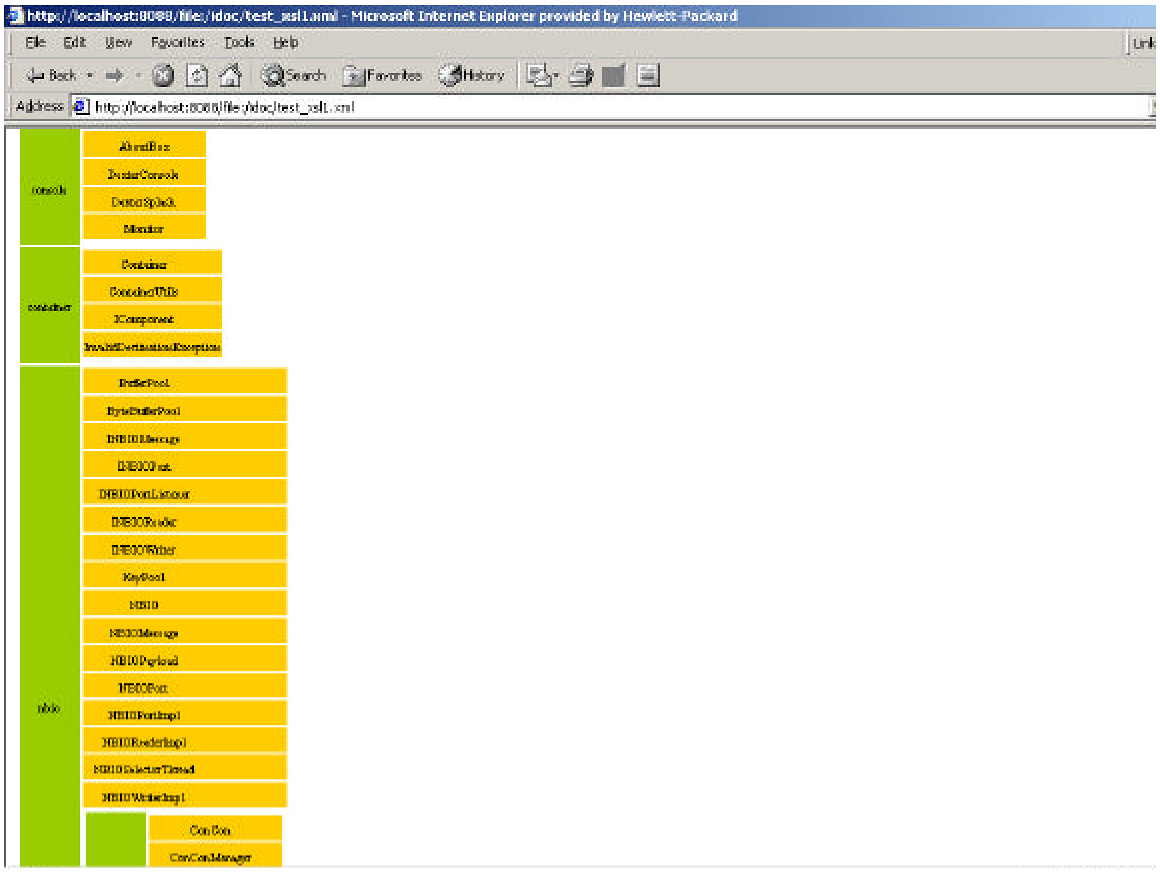


Figure 33: The console, container and nbio packages

Figure 34, contains the packages that make up the bulk of the high-level components inside the container and which were highlighted in Figure 6. The *cache* package is self-explanatory, the *core* package contains classes related to monitoring and the InstructionPump, the *id* package contains the Instruction Decoder and supporting classes. The security and system packages provide for logging, path name resolution and access control. The *uio* package contains the code relating to the Universal Server component.

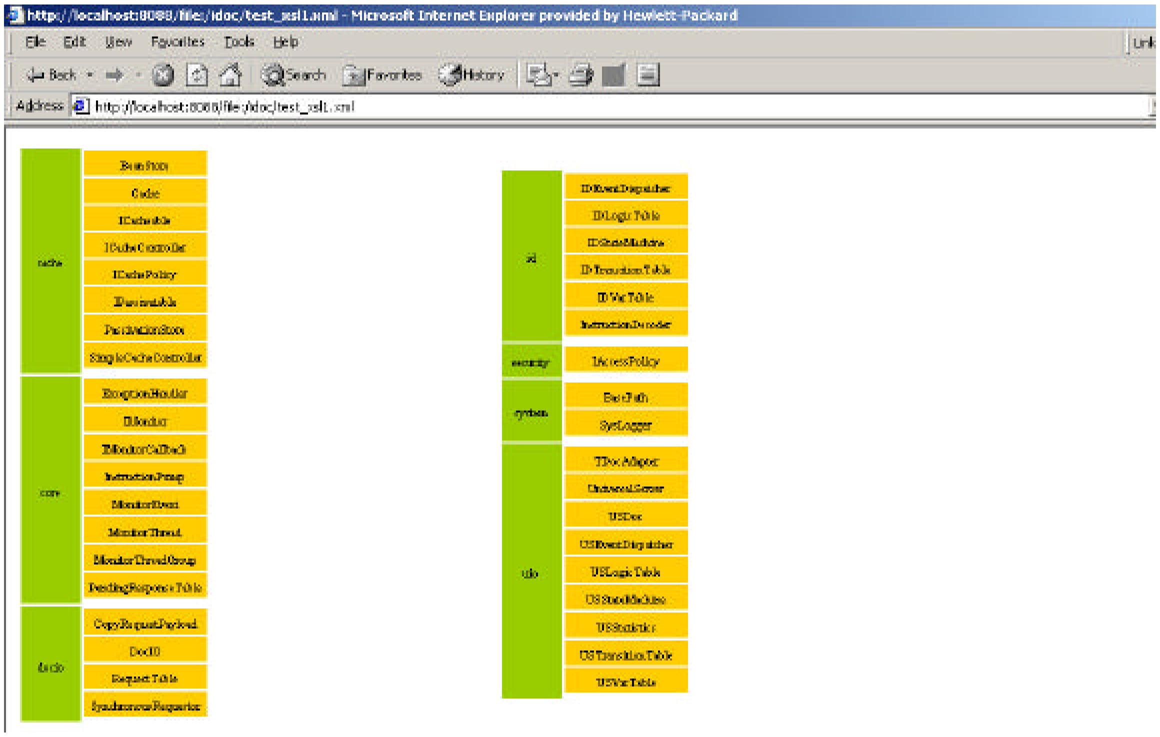


Figure 34: The cache, core, docio, id, security, system and uio packages

Most of the high-level components are built up from several basic subclasses. Figure 35 shows two packages, the *queue* package and the *statmach* package, which contain several of these basic sub-classes. Three types of queue are used internally in the Dexter XAP. These are a *Simple* queue, a *Priority* queue and a *Correlating* queue (supports asynchronous callbacks). The *statmach* package contains the primitives for building state-machines. These are used in the Instruction-Decoder, and the Universal-Server to implement the internal protocols that exist between these components and the DocIO.

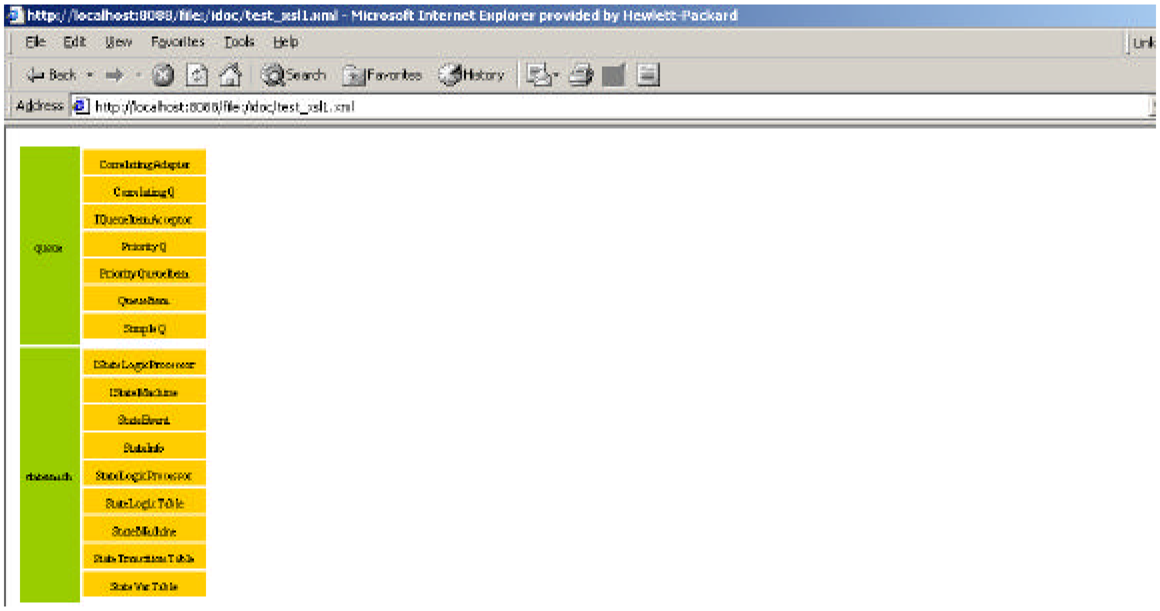


Figure 35: The queue and statemach packages

Figure 36, shows the *thread* and *transport* packages which contain additional basic subclasses. The *thread* package contains the basic thread types for implementing the DocIO, the Universal-Server, the Instruction-Pump and the Instruction-Decoder. The *transport* package holds all the supported transports managed by the Transport Manager and the Transport Manager class itself.

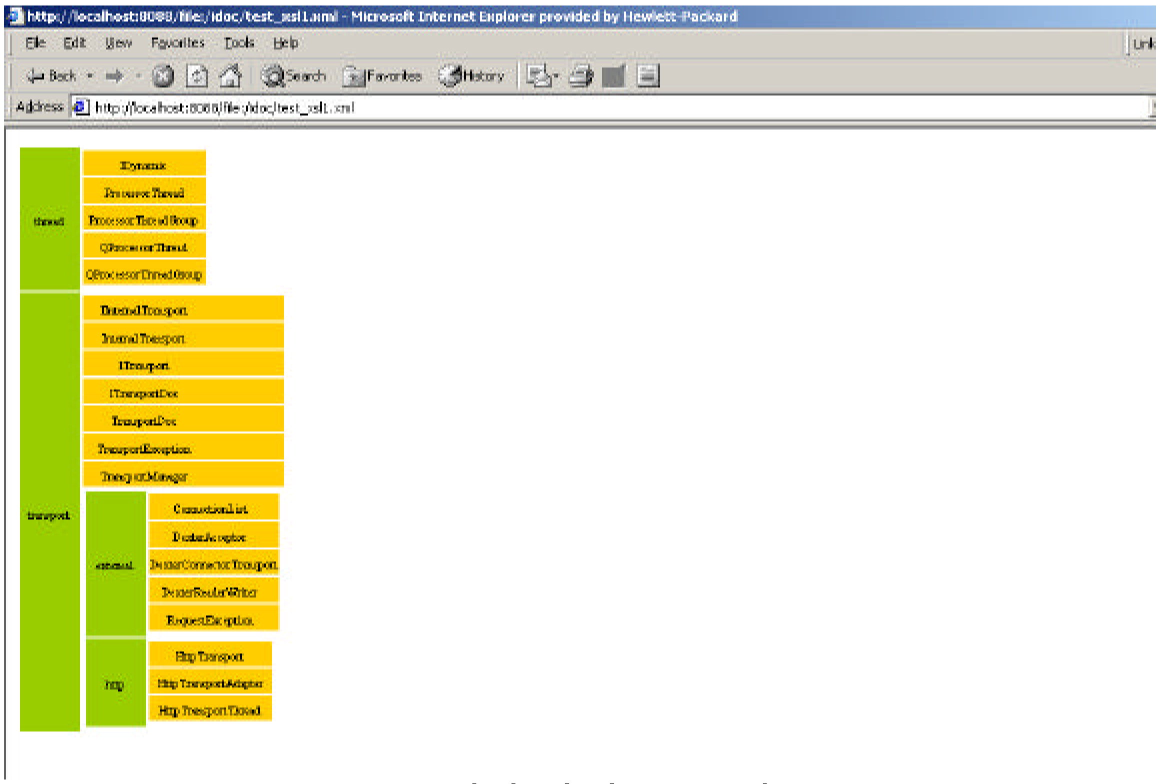


Figure 36: The thread and transport packages

The *uclient* package (Figure 37) contains all the Document Accessors and helper classes.

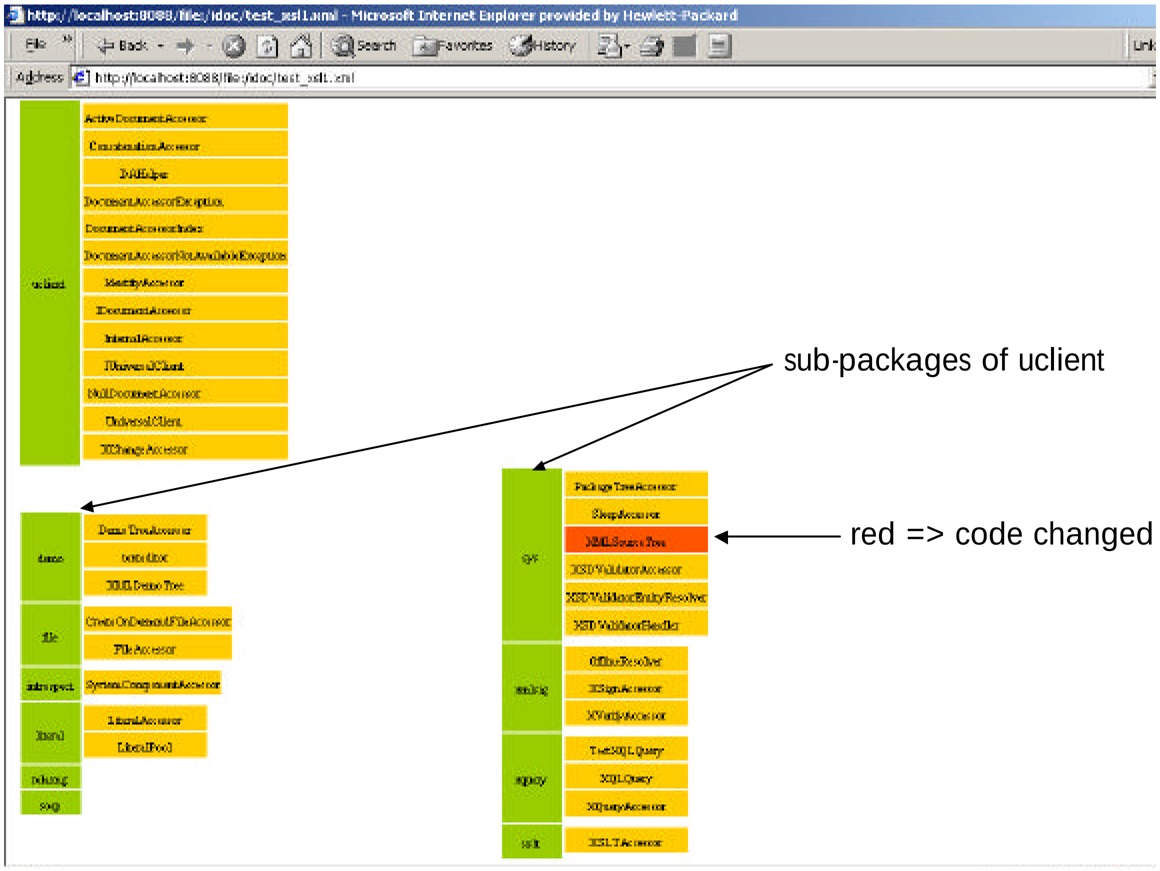


Figure 37: The uclient package

The *utils* package (Figure 38) provides a variety of classes to handle resource locking, XML manipulation, and URI creation etc.…

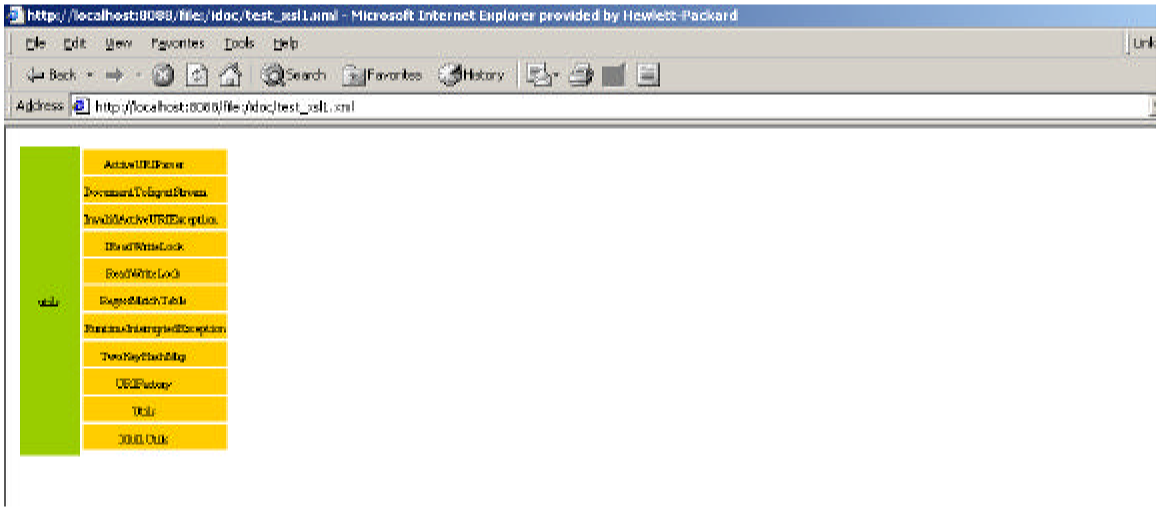

http://localhost:8088/file:/idoc/test_xsl1.xml - Microsoft Internet Explorer provided by Hewlett-Packard
File
Edit
View
Favorites
Tools
Help
Links
Back
Search
Favorites
History
Address
http://localhost:8088/file:/idoc/test_xslt.xml
utils
ActiveURIParser
DocumentToInputStream
InvalidActiveURIException
IReadWriteLock
ReadWriteLock
RegexMatchTable
RuntimeInterruptedException
TwoKeyHashMap
URIFactory
Utils
XMLUtils

Figure 38: The utils packages

Finally the XMLBean package (Figure 39) contains all the XMLBean interfaces as described in section 5.2. It also includes a set of sub-packages containing all the XMLBean subclasses.

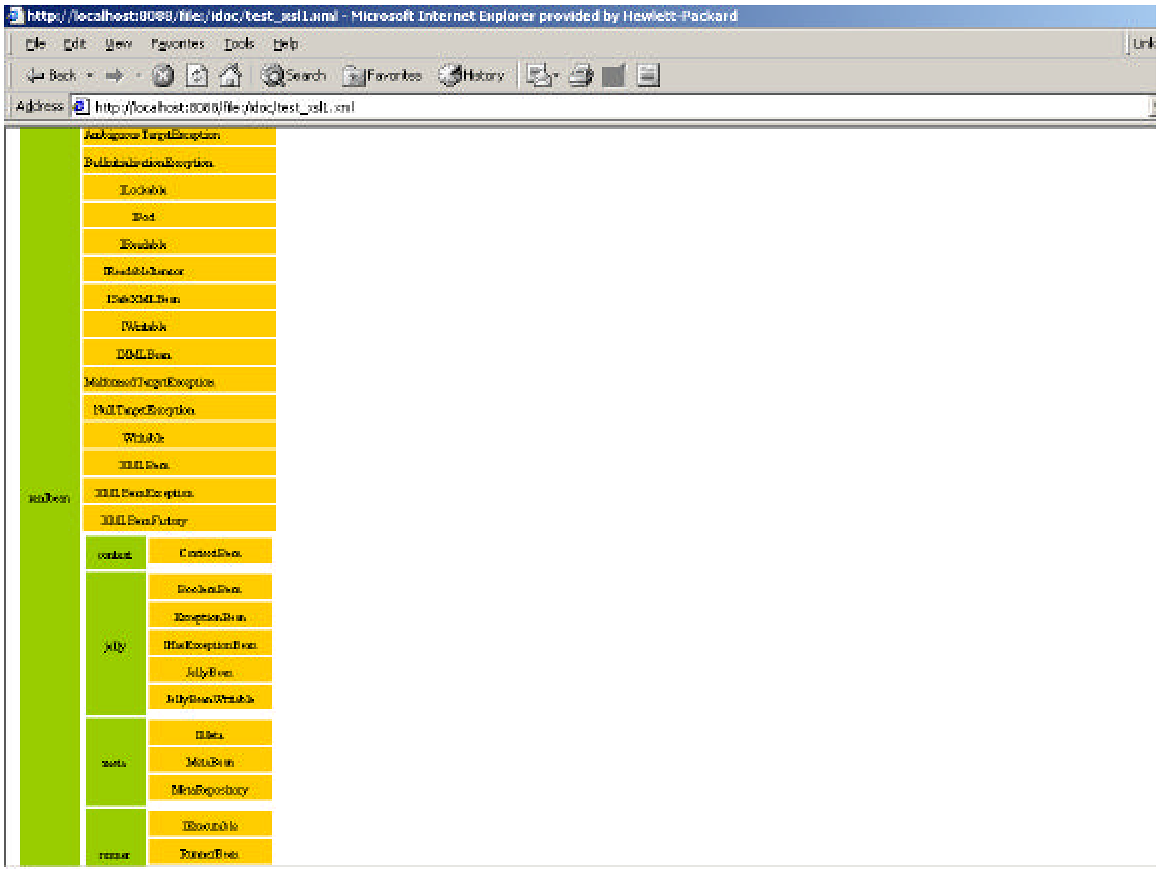


Figure 39: The xmlbean package

11 Appendix B: The XMLBean and IDocAccessor Interfaces

We give here abridged JavaDoc for selected interfaces in the Dexter XAP's XMLBean system and IDocAccessor.

11.1 IReadable

An interface to read the XML document encapsulated by the XMLBean without resorting to DOM API calls.

Method Summary	
`java.lang.String`	**get**`(java.lang.String aXPath, boolean aTrim)` Returns the value from one node within the document.
`org.w3c.dom.Document`	**getFragment**`(java.lang.String aXPath)`
`java.util.Map`	**getMulti**`(java.lang.String aXPath, boolean aTrim)` Returns a Map of Simple XPath of node to value for all nodes found.
`IReadableIterator`	**getReadableIterator**`(java.lang.String aXPath)` Similar to getMulti it evaluates an XPath to a possible set of result nodes.
`org.w3c.dom.Document`	**toDOM**`()` Returns a DOM representation of the document contained in this bean.
`java.lang.String`	**toString**`(boolean aIndent)` Returns a debug representation of the contained document.
`java.lang.String`	**toString**`(java.lang.String aXPath, boolean aIndent)`

11.2 IWritable

An interface to write to an XML document encapsulated by the XMLBean.

Method Summary	
`void`	**addMultiPath**`(java.lang.String aContext, java.lang.String aNewPath, java.lang.String aValue)` Creates new elements, attributes, and text nodes by building from zero or more context nodes down and then creating either an attribute or text nodes at the bottom.
`void`	**addPath**`(java.lang.String aContext, java.lang.String aNewPath,`

	java.lang.String aValue) Creates new elements, attributes, and text nodes by building from a context node down and then creating either an attribute or text nodes at the bottom.
void	**addPI**(java.lang.String aXPath, java.lang.String aTarget, java.lang.String aValue) Adds a processing instruction to the document below the location node given by aXPath.
void	**copy**(IReadable aSrc) Copies the given IReadable to this Writable
void	**copy**(IReadable aSrc, java.lang.String aSrcPath, java.lang.String aDstPath) Copies the given IReadable fragment to this Writable at the given location
void	**deleteMultiPath**(java.lang.String aPath) Deletes zero or more elements or attribute defined by an XPath expression.
void	**deletePath**(java.lang.String aPath) Deletes a single element or attribute defined by an XPath expression.
void	**insertPath**(java.lang.String aContext, java.lang.String aNewPath, java.lang.String aValue) Creates new elements, attributes, and text nodes by building from a context node down and then creating either an attribute or text nodes at the bottom.
void	**set**(java.lang.String aXPath, java.lang.String aValue) Sets the value of either an attribute or the text below an element.

Methods inherited from interface com.hp.hpl.dexter.xmlbean.IReadable
get, getFragment, getMulti, getReadableIterator, toDOM, toString, toString

11.3 IExecutable

The interface which a class must implement in order to be executed within the Dexter Container. In the Dexter prototype, an XMLBean called the RunnerBean implemented this interface. The idoc workflow documents are encapsulated by RunnerBeans.

Method Summary	
java.lang.String	**getHumanReadableCurrentInstruction**(ContextBean cxt) Returns a string with details about the current instruction.
java.lang.String	**getInstructionType**(ContextBean cxt) Requests what the type of instruction to perform.
java.net.URI	**getOperand**(ContextBean cxt)

	Requests the URI of the operand for the current instruction.
java.lang.String	**getOperandType**(ContextBean cxt) Requests the source type of the operand
java.net.URI	**getOperator**(ContextBean cxt) Requests the URI of the operator document.
java.lang.String	**getOperatorType**(ContextBean cxt) Requests the source type of the operator
java.net.URI	**getParameter**(ContextBean cxt) Requests the URI of any parameter document.
java.lang.String	**getParameterType**(ContextBean cxt) Requests the source type of the parameter
java.net.URI	**getTarget**(ContextBean cxt) Request the URI of the target to write the results of the operation
java.lang.String	**getTargetType**(ContextBean cxt) Requests the source type of the target.
java.lang.Boolean	**isAsynchronous**(ContextBean cxt) Tests whether the instruction can be performed asynchronously.
short	**nextInstruction**(ContextBean cxt) Requests the instruction pointer be moved to the next instruction.

11.4 ILockable

An interface for acquiring and releasing locks on XMLBeans. Locks are held against the thread which asks for them.

Method Summary	
IReadable	**getReadable**() Acquire a lock to read an XMLBean and return an interface to allow reading.
IReadable	**getReadable**(java.lang.Object aToken, long aTimeout) Acquire a lock (using a token) to read an XMLBean and return an interface to allow reading.
IWritable	**getWritable**() Acquire a lock to write to an XMLBean and return an interface to allow writing.
IWritable	**getWritable**(java.lang.Object aToken, long aTimeout) Acquire a lock (using a token) to write to an XMLBean and return an interface to allow writing.
void	**releaseLock**() Releases any held locks on this bean by the current thread.
void	**releaseLock**(java.lang.Object aToken)

	Releases any held locks on this bean by the given token.

11.5 ICacheable

An interface which classes must implement in order to be put into the container cache. The Cache controller uses this interface for managing the lifecycle of a cached object.

Method Summary	
`long`	**getAge**`()` Return the time since the object was created.
`int`	**getFixCount**`()` Returns the number of clients fixing the object into the cache.
`long`	**getMeanTimeBetweenAccesses**`()` The avaerage number of milliseconds between accesses to the object.
`long`	**getTimeSinceLastAccess**`()` The number of milliseconds since the last access of the object.
`void`	**incrementFixCount**`(int aIncrement)` Adjusts the fix count relative to its current value.
`boolean`	**isDirty**`()` Return true if the object has been modified but not persisted.

11.6 IPod

The interface for XMLBeans to allow them to contain references to other XMLBeans.

Field Summary	
`static java.lang.String`	**ROLE_META** constant for the role of a Meta document- needed for all beans within dexter.

Method Summary	
`void`	**addBean**`(java.lang.String aRoleName, IXMLBean aBean)` Makes the given bean contained by the current bean.
`IXMLBean`	**getBean**`(java.lang.String aRoleName)` Returns a bean fulfilling a particular role.
`java.util.Iterator`	**getRoles**`()` Returns an iterator over the names of all the roles.

void	**removeBean**(java.lang.String aRoleName) Stops the bean with the named role being contained by this bean.

11.7 IPassivatable

Interface for objects that can be passivated to and activated from the PassivationStore.

Method Summary	
void	**activate**(java.io.Reader aReader, Cache aCache) Activate the this object with data from the reader.
void	**passivate**(java.io.Writer aWriter) Passivate this object to the given writer

11.8 IDocumentAccessor

A interface capable of allowing access to the document for a given URI independent of its implementation or location.

Method Summary	
boolean	**checkSupport**(int aSupportFlags) Check to see if support exists for a given operation.
void	**deleteDocument**(java.net.URI aURI) Deletes the given URI such that it should not exist in future.
boolean	**documentExists**(java.net.URI aURI) Check to see if the given URI exists.
void	**sinkDocument**(java.net.URI aURI, java.io.Reader aDocument) Updates the given URI with data from the given reader.
org.w3c.dom.Document	**sourceDocumentByDOM**(java.net.URI aURI) Creates a DOM document for the data for a given URI.
java.io.Reader	**sourceDocumentByStream**(java.net.URI aURI) Creates a reader for the data for a given URI.

12 References

[1] ebXML. See http://www.ebxml.org/
[2] Interactive Financial exchange Forum. See http://www.ifxforum.org/ifxforum.org/index.cfm
[3] XML Path Language (XPath). See http://www.w3.org/TR/xpath
[4] XML Pointer, XML Link. See http://www.w3.org/XML/Linking
[5] XML Query. See http://www.w3.org/XML/Query
[6] XSL Transformations. See http://www.w3.org/TR/xslt
[7] Web Services Description Language (WSDL) http://www.w3.org/TR/wsdl
[8] XSL Formatting Objects. See http://www.w3.org/TR/xsl/slice6.html - fo-section
[9] XML-Signature Syntax and Processing. See http://www.w3.org/TR/xmldsig-core/
[10] XML Schema. See http://www.w3.org/XML/Schema
[11] RelaxNG. See http://www.oasis-open.org/committees/relax-ng/
[12] XUpdate. See http://www.xmldb.org/xupdate/
[13] http://www.rosettanet.org/
[14] Business Process Execution Language for Web Services. See http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/
[15] Common Gateway Interface. See http://hoohoo.ncsa.uiuc.edu/cgi/overview.html
[16] Microsoft XDocs. See:
http://www.microsoft.com/presspass/features/2002/Oct02/10-09OfficeFamily.asp
[17] Web Services for Interactive Applications. See http://www.oasis-open.org/committees/wsia/
[18] XForms. See http://www.w3.org/TR/2002/WD-xforms-20020118/
[19] Web Services for Remote Portals. See http://www.oasis-open.org/committees/wsrp/
[20] XML Encryption. See http://www.w3.org/Encryption/2001/
[21] XML Access Control Markup Language. See http://www.oasis-open.org/committees/xacml/
[22] XML Information Set. See http://www.w3.org/TR/xml-infoset/
[23] Schmidt, D.C.; Stal, M.; Rohnert, H.; Buschmann, F. *Pattern-Oriented Software Architecture*. Wiley, 2000.
[24] Business Transactions Protocol. See https://www.oasis-open.org/committees/business-transactions/
[25] User Interface Markup Language. See http://www.uiml.org/index.php
[26] XML Pipeline Definition Language. See http://www.w3.org/TR/xml-pipeline/
[27] Web Service Choreography Interface. See http://www.w3.org/TR/wsci/
[28] Shegalov, G.; Gillmann, M.; and Weikum, G. *XML-enabled workflow management for e-services across heterogeneous platforms*. The VLDB Journal, 2001. 10:91-103. Available online at http://citeseer.nj.nec.com/shegalov01xmlenabled.html
[29] XCool XML Query Language Implementation http://xcool.sourceforge.net/
[30] XSL-T Processing Engine http://xml.apache.org/xalan-j/