

Developing Applications with XML Documents, Document Transformations and Software Components

J.L. Sierra, Baltasar Fernández-Manjón, Alfredo Fernández-Valmayor, Antonio Navarro

Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Avda.
Complutense S/N, 28040 Madrid, Spain
{jlsierra, balta, alfredo, anavarro}@sip.ucm.es

Abstract. This paper describes the DTC (Documents, Transformations and Components) approach to XML-based development of software applications. According to this approach the content of an application, and key aspects of its structure and behavior, are represented in terms of XML documents. In DTC, software development is conceived as the process that gives operational support to these documents. DTC also encourages the organization of software in terms of reusable components, each of them able to process a specific markup language. Because documents and components could have been reused from pre-existing repositories, we use document transformations to fit them together. In this paper, we describe the DTC approach, illustrating its application in a case study. Because DTC encourages the explicit separation between application's content, software descriptions and operational support, we improve maintainability and reuse at both information and software levels.

1 Introduction

Generalized markup languages, such as SGML (*Standard Generalized Markup Language*) [4] or XML (*eXtensible Markup Language*) [11], can be used to make explicit the structure of a document. These languages allow the definition of a markup vocabulary and a set of grammatical rules to properly combine such vocabulary. Because it is possible to select the most suitable document grammar and vocabulary for each domain, the use of generalized markup languages avoid the rigidity that a single data model or encoding formalism imposes to the domain modeler. However, markup languages only describe how the information is structured. The use of a marked document for performing a particular task requires, at the end, the existence of an external program giving operational meaning to document structures. Building such a program can be a complex software development activity. Although the use of general purpose software, such as markup parsers and editors, can be helpful at lowering the overall development complexity, it does not solve the most critical part of the problem: the construction of the domain dependent semantics.

We think that a smart use of componentware technology can help to fill the existing gap between syntax specification of a structured document and the desired operational semantic for the intended use of this document. We consider that the key

idea is to devise components specialized in the interpretation of particular markup languages. In addition, such languages would not be compromised with any specific domain. This practice would make components supporting such languages reusable for different purposes. In this way, we need mechanisms to bring different reusable software components together and, if we also reuse existing documents, to integrate these documents in the overall application. Document transformations provide us with these mechanisms. We have put together these key ideas in our approach for developing XML-based applications. We have called this approach DTC, from *structured Documents*, *document Transformations* and *software Components*. In this paper we describe our preliminary experiences with DTC: Section 2 outlines the DTC approach itself, section 3 describes how DTC is applied in a case study and, finally, section 4 outlines some conclusions and future work.

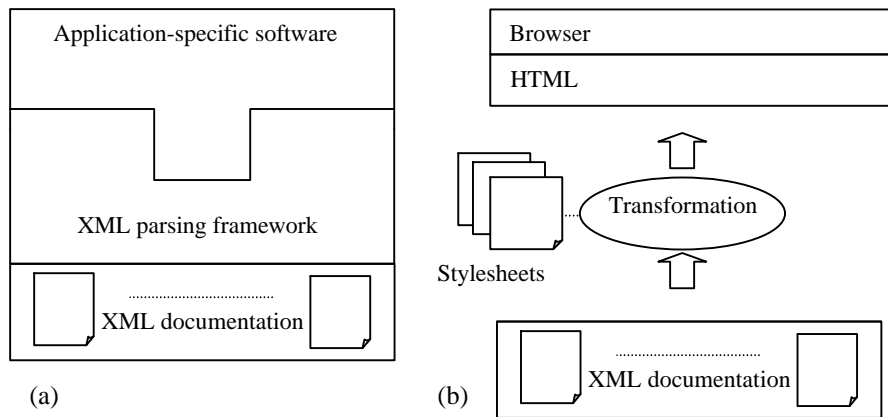


Fig. 1. (a) Typical full-custom structure for a XML-based application. (b) Typical strategy followed in XML-based web publication

2 The DTC Approach

Building an (executable) XML-based application requires the provision of software for processing one or several classes of XML documents. Fig.1a outlines a typical *full custom* structure for such applications [7][8]. The applications rely on the use of general-purpose parsing/generation frameworks [2][10] connected with application specific software. The main weakness of this organization is the coupling between this application specific software and document structure (that hardly enables reusing this software in different applications) and the costs associated with its development from scratch.

A different approach, that works for specialized uses such as XML-based web publication, enables to reuse pre-existing software, such as web servers and web browsers. The publication task is subsequently intended as a *transformation step* from

the source XML documentation to a presentation format (in case of web publication, HTML –HyperText Markup Language- [12]) for which processing software is already available (Fig.1b). Consequently, development cost dramatically decreases. Unfortunately the feasibility of this approach strongly depends on being able to reformulate the task at hand as a publishing one. Of course it will be argued that the target presentation environment always could be extended with the loose processing capabilities (for instance, by means of scripting). But this leads again to the need of providing an important part of the domain-specific software from scratch.

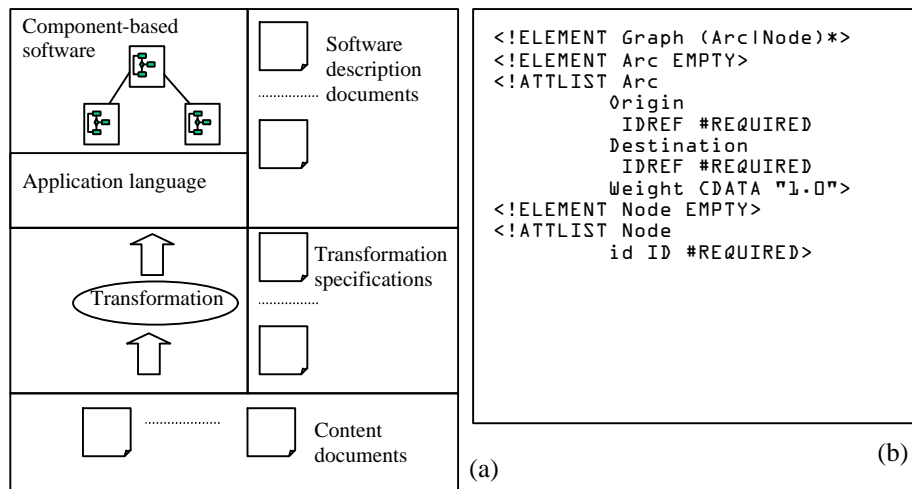


Fig. 2. (a) Structure of a DTC application. (b) A markup language for representing weighted directed graphs. This DTD can be associated with a software component for processing DTD-conforming documents

DTC approach proposes an intermediate solution for developing XML-based applications. Instead of recurrently building software from scratch, or, in the other side, trying to look for the universal language and browser, DTC suggests the use of software components specialized in processing specific classes of XML documents. In this way, each component is tightly associated with a markup language and can be understood as giving operational support for such a language. From this viewpoint, building a DTC application requires, on one hand, to provide the content to be managed by the application in XML terms, and, on the other hand, to properly combine a set of components giving computational support to the end-application. Such combination process leads to a component-based computational artifact able to process information conforming the languages associated to some of their components. In this way, having the application ready to be executed will also imply a transformational step from the original XML documentation to the documents required by the components. Such transformations can, in their turn, be described using suitable transformation specification documents (for instance, in terms of

XSLT, *eXtensible Stylesheets Language Transformations* [13]). Finally, combination itself can make use of higher-level software components oriented to mastering other components. Such *combinator components* can also be parameterized in terms of suitable markup descriptions. Fig. 2a summarizes the general structure of a DTC application. Next subsections detail the most relevant features of the approach.

2.1 The Content Documents

In a DTC application the contents are structured in terms of marked documents that are jointly named *content documents*. Attending to their purpose, it is possible to distinguish between two different kinds of these documents. *Domain content documents* contain domain specific information that could be reused across different applications (for instance, a dictionary, a botanical glossary, etc). On the other hand, there are *application dependent content documents* with a clear meaning only inside a specific application (for instance, a document enclosing metrical information of a roadmap to be used in a graphical presentation). Application dependent content documents are mainly oriented to *complete* the information provided by reusable domain content documents.

2.2 The Application Software, the Application Language and the Software Description Documents

The applications built according DTC are component-based: they are built by means of the selection/construction, configuration and assembling of software components [9]. There are three main kinds of components in a DTC application: *markup interpreters*, *primitive facilities* and *combinators*.

Markup interpreters have a great relevance inside the DTC approach. They are oriented to process a specific markup language. A good example of these interpreters is a component for processing weighted directed graphs described in terms of the XML DTD (*Document Type Definition*) of Fig. 2b. This component can give several uses to the graphs represented in terms of the language defined by that DTD (eg. searching the minimum-cost path, computing the minimum spanning tree, etc). Content processing mainly relies on the set of markup interpreters included in a DTC application. In this way, the markup languages associated with these components jointly define the *application language*: the language in which all the content processed by the application must be finally translated.

Primitive facilities are components that carry on basic functionality in the final application. For instance, basic GUI components (buttons, labels, menus, etc.) fit inside this category. Another class of primitive facilities are given by general XML processing components such as query or transformation engines that can help to assemble together different independently deployed, reusable components when they interchange information in XML terms. Finally notice that primitive facilities can have an associated markup language for enabling its configuration. Documentation required by such components have nothing to do with content processing. These

documents, together with the other documents used for describing structure or behaviour of the application software, are named *software description documents*.

Combinators make possible to set up the way in which other components behave and interact. So, DTC does not commit itself with a pre-established combination strategy. Specific strategies are explicitly introduced by appropriated combinators. Some examples of combinators are typical *GUI containers*, such as those included in the AWT or Swing Java APIs. These combinators can come up with a markup language for configuring things such as look and feel, layout politics, etc. Other kind of combinators will be devoted to control the behaviour of simpler components. Good examples of these *controllers* are components supporting tailored scripting languages, control formalisms such as state machines or Petri nets, or event-driven component interconnection languages. Like primitive facilities, many of these components become with their associated configuration markup languages. In this way, their use in an application requires the provision of the appropriated software description documents conforming these languages.

2.3 Putting All Together

Document transformations are specifications for deriving *result documents* from *source documents* (or, more precisely, from parse trees of source documents to parse trees of result documents [6]). Tree filter programming languages, such as XSLT or, in the SGML world, DSSSL (*Document Style Semantics and Specification Language*) [5] are normally used to specify such transformations. Because information produced and consumed by DTC software components is XML-structured, transformations enables the adaptation of these information flows between reusable components. In addition, transformations are used in DTC for mapping content documents into the application language.

When the software of a DTC application is built as described in the previous subsection, an application language is automatically induced. Such a language can be understood as the composition of all the languages for the markup interpreters included in the application. Of course it could be possible to directly provide the application with documents written in this language. But doing it so have several disadvantages: (i) it prevents to reuse pre-existing domain documentation, (ii) application languages derived from those supported by reusable components could be difficult to understand for domain experts providing the contents, (iii) because each component can require different views of the same information, direct provision of contents in application language terms can lead to provide similar information multiple times, and (iv) application languages usually are *task oriented*; that is, information provided in terms of these languages will be hardly usable for any other purpose (it is the same reason because information is encouraged to be represented in tailored XML languages instead of being directly encoded in a task – oriented one, such as HTML!). For all the reasons above, DTC encourages to separate content and application languages, and to use document transformations for mapping the content of the application into the application language.

3 A Case Study: The Subway Application

In this section we present a case study of applying DTC in the development of a non-trivial application. The application provides an interactive graphical interface to find the best route between any two given stations in a subway network. We have instantiated the application in the subway network of Madrid (Spain). We used Java as implementation technology, together with the Oracle Java Libraries for XML parsing and XSLT support [3].

```
<!ELEMENT SubwayNetwork
(Stations,Corridors?,Lines)>
<!ELEMENT Lines (Line)+>
<!ELEMENT Line (Schedulers,Links)+>
<!ATTLIST Line id ID #REQUIRED>
<!ELEMENT Schedulers (Scheduler)+>
<!ELEMENT Scheduler EMPTY>
<!ATTLIST Scheduler
StartTime CDATA #REQUIRED
EndTime CDATA #REQUIRED
Frequency CDATA #REQUIRED>
<!ELEMENT Links (Link)+>
<!ELEMENT Link EMPTY>
<!ATTLIST Link
OriginStation
IDREF #REQUIRED
DestinationStation
IDREF #REQUIRED
Distance
CDATA #REQUIRED
Speed
CDATA #REQUIRED>
<!ELEMENT Corridors (Corridor)+>
<!ELEMENT Corridor EMPTY>
<!ATTLIST Corridor
id ID #REQUIRED
OriginStation
IDREF #REQUIRED
DestinationStation
IDREF #REQUIRED
TraversingTime
CDATA #REQUIRED>
<!ELEMENT Stations (Station)+>
<!ELEMENT Station
(Accesses,Tracks,Times) >
<!ATTLIST Station id ID #REQUIRED>
<!ELEMENT Accesses (Access)+ >
<!ELEMENT Access (#PCDATA) >
<!ATTLIST Access id ID #REQUIRED>
<!ELEMENT Tracks (Track)+ >
<!ELEMENT Track EMPTY >
<!ATTLIST Track
id ID #REQUIRED
Line IDREF #REQUIRED
Direction IDREF #REQUIRED >
<!ELEMENT Times (AscentTime |
DescentTime |
TransferTime)+ >
<!ELEMENT AscentTime EMPTY>
<!ATTLIST AscentTime
Track IDREF #REQUIRED
Access IDREF #REQUIRED
Time CDATA #REQUIRED>
<!ELEMENT DescentTime EMPTY>
<!ATTLIST DescentTime
Access IDREF #REQUIRED
Track IDREF #REQUIRED
Time CDATA #REQUIRED>
<!ELEMENT TransferTime EMPTY>
<!ATTLIST TransferTime
originTrack IDREF #REQUIRED
DestinationTrack
IDREF #REQUIRED
Time CDATA #REQUIRED>
```

Fig. 3. DTD for representing information about a subway network

3.1 The Application Contents

The subway application includes: (i) a domain content document, marked according to the DTD of Fig.3, with information about the subway structure (stations, corridors, accesses, subway lines, etc.) and timing (schedulers, trajectory times between different points of a station, average speed of each line, etc.), and (ii) an application dependent content document with geometrical information for rendering the subway map. Notice that the direct provision of all this information can be a tedious work. This work can be avoided by building and using special-purpose edition tools for the

required information. Section 4 will suggest how DTC can be extended for coping with these authoring activities.

<pre> <subwayApplication> <mainWindow scalable="no" title= "subway route finder"/> <mainPanel background="pink"> <row> <component>map</component> <component>lateralPanel </component> </row> <row> <component>controlLabel </component> </row> </mainPanel> ... <automata> <init state="init"/> <state id="init"> <action> mainWindow>visualize(); originLabel>changeText (text = ""); destinationLabel>changeText (text = ""); controlLabel>changeText (text = "Select origin station"); </action> <transition state="selectingOrigin"/> </state> ... </automata> </subwayApplication> </pre>	<pre> <xsl:template match="SubwayNetwork"> <graph> <xsl:apply-templates/> </graph> </xsl:template> <xsl:template match= "Station Track"> <node id="{@id[1]}" /> <xsl:apply-templates/> </xsl:template> <xsl:template match="Access"> <node id="{@id[1]}" /> <arc origin="{@id[1]}" destination= "{../../@id[1]}" cost="0"/> <arc origin="{../../@id[1]}" destination="{@id[1]}" cost="0"/> </xsl:template> <xsl:template match="AscentTime"> <arc origin="{@Track[1]}" destination="{@Access[1]}" cost="{@Time[1]}" /> </xsl:template> <xsl:template match="DescentTime"> <arc origin="{@Access[1]}" destination="{@Track[1]}" cost="{@Time[1]}" /> </xsl:template> ... </pre>
(a)	(b)

Fig. 4. (a) Part of the software description document for the subway application. (b) Part of the XSLT specification document for transforming the subway description in a weighted graph

3.2 The Application Software, the Application Language and the Software Description Documents

The application software is built using components for each one of the three categories introduced in subsection 2.2. In the first category, two markup interpreters are used: *Diagram*, giving support for a simple language that enables the description of 2D diagrams made of circles, straight line connections and text labels, and *Graph*, giving operational support for a weighted directed graph description language, similar to that of Fig. 2b. In the second category, primitive facilities, we use basic GUI

facilities such as buttons and labels. In addition we use a simple *Map* transformation engine for translating list of nodes (given in the language of the Graph component) into lists of stations. For managing these lists (in order to visualize them in the diagram representation of the subway map) we use a generic *XML processor* component, allowing the manipulation of documents in terms of their DOM (Document Object Model) trees [10]. Finally for the third category, combinators, we have used typical GUI containers, and for describing control, an *Automata* controller that gives support for a state-transition oriented formalism.

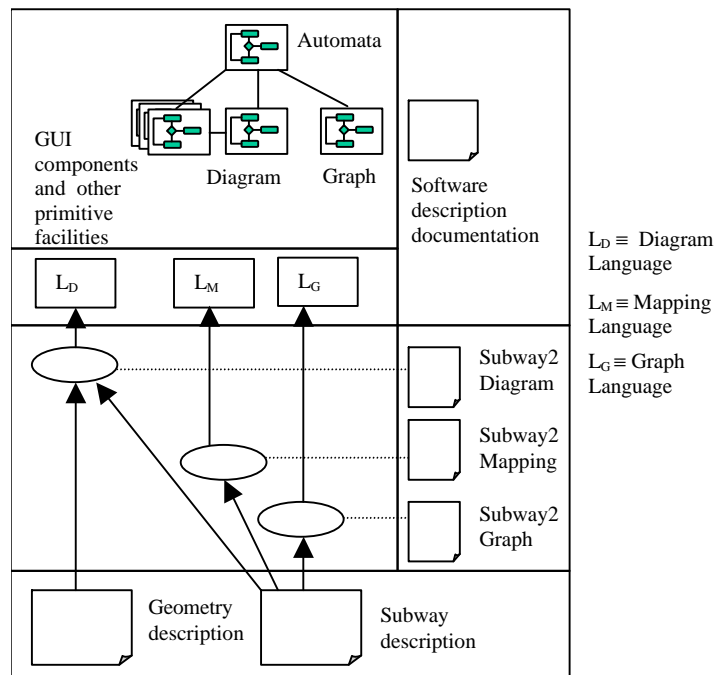


Fig. 5. DTC structure of the subway application

Notice that our application language is mainly given by the languages associated with the two markup interpreters. They support two different views of the subway network: a view as a graph, and a view as a diagram. Because a relation between these two views is required (for visualizing routes expressed as lists of nodes, in terms of the graph language), the map used for the *Map* transformation engine is also included in this language.

Finally, the software description documentation for the other components must be provided. To improve maintainability we group all this information in a single document (Fig. 4a). From this document, individual software descriptions for each component are derived using simple query and transformation steps.

3.3 Putting All Together

Having the content documents and the computational support for the application, only rests to put all together. For doing it we need to give the transformations from the content languages to the application one. A transformation enables the diagram view of the subway network to be generated. Such a transformation takes both the domain and the presentational documents as sources. A second transformation is used for generating the graph view (Fig. 4b shows a fragment of an XSLT filter for this transformation). Finally, another transformation is used for generating the document that encloses the relation between the two views.

Fig. 5 sketches the structure of the final application. Fig. 6 shows a snapshot of the application itself.



Fig. 6. Screenshot of the subway's route finder application built using the DTC approach

4 Conclusions and Future Work

We advance that the DTC approach improves the maintainability of applications, because the explicit separation between content and computational machinery and because the representation of information as human readable and editable documents [1]. Many of the changes and updates in the application are at the document level with no programming effort. In addition, the DTC approach also takes advantage of the component-based software construction modularity for easing update and

maintenance. The DTC approach also encourages reusability at different levels. Domain content documents and DTDs can be reused for multiple purposes. Software components can also be reused in the construction of different applications. Finally application software can be reused for building new applications in similar domains. Document transformations are used as the basic glue for enabling both reusable documentation and reusable software components to work together.

The most relevant shortcomings of the DTC approach, in its current state, are the complexity of managing efficiently the different sorts of information (domain, application and transformation specification documents, application software description, etc.) and the authoring of the application content documents. The complexity of the DTC process can be lowered with a suitable automation. Currently we have developed a batch environment for doing all this work, but we plan to develop a graphical tool for supporting the DTC process. In order to improve DTC with authoring facilities, the same component-oriented and information and software separation ideas underlying the approach could be applied. Currently we are working on an extension of DTC oriented to the generation of domain dependent document editors. The idea is to derive specialized editors from reusable DTC components (extended to support editing capabilities). Because such components must generate structured documents according to their supported languages, *inverse transformations* are needed for generating domain content documents from documents in the application language. We refer to this approach as the *inverse DTC*.

References

1. Fernandez-Manjon, B., Navarro, A., Cigarran, J., Fernandez-Valmayor, A.: Using Standard Mark-up in the Design and Development of Web Educational Software. Teleteaching'98, Proceedings of the 15th IFIP World Computer Congress (1998) 303-312
2. <http://www.megginson.com/SAX/>
3. <http://technet.oracle.com/>
4. International Standards Organization.: Standard Generalized Markup Language (SGML). ISO/IEC IS 8879 (1986)
5. International Standards Organization.: Document Style Semantics and Specification Language (DSSSL). ISO/IEC 10179 (1996)
6. Kuikka, E., Penttonen, M.: Transformation of Structured Documents. Tech. Report CS-95-46, University of Waterloo (1995)
7. Maruyama,H.,Tamura,K.,Uramoto,N.: XML and Java - Developing Web Applications. Addison Wesley (1999)
8. St.Laurent,S.,Cerami,E.: Building XML Applications. Osborne Mc Graw-Hill (1999)
9. Szyperski, C.: Component Software - Beyond Object-Oriented Programming. Addison Wesley (1998)
- 10.W3C Candidate Recommendation.: Document Object Model (DOM) Level 2 Specification Version 1.0. <http://www.w3.org/TR/DOM-Level-2> (2000)
- 11.W3C Recommendation.: Extensible Markup Language (XML) 1.0. <http://www.w3.org/XML> (1998)
- 12.W3C Recommendation.: HTML 4.01 Specification. <http://www.w3.org/TR/html401> (1999)
- 13.W3C Recommendation.: XSL Transformations (XSLT) Version 1.0. <http://www.w3.org/TR/xslt> (1999)