DEVELOPING A FLEXIBLE ELECTRONIC PATIENT RECORD AS A WEB OF ACTIVE DOCUMENTS

Federico Cabitza and Giovanni Zorzato
Università degli Studi di Milano-Bicocca, Viale Sarca 336, Milan, Italy

Keywords: Active documents, Webs of active documents, Electronic patient record, Didgets.

Abstract: In this paper, we discuss the architecture of WOAD, a design-oriented framework that we proposed to enact a bottom-up and document-centered approach to the development of Electronic Patient Records. We provide the essential elements of WOAD: the concept of Active Document, Didget, Template and Mechanism. Then we summarize the observational studies that inspired its development and that gave the preliminary user feedback for its validation by means of the deployment of ProDoc, a WOAD-compliant patient record. We then illustrate the core implementation details of the WOAD architecture, as it has been deployed in ProDoc.

1 MOTIVATIONS AND BACKGROUND

In the last five years, our research has focused on the analysis of the Electronic Patient Records (EPR) used in specific hospitals of our region and on the design of innovative EPRs that could improve user experience, compliance to hospital and regional healthcare policies, data quality and patient safety. To gain the necessary knowledge on the EPRs used in the considered settings, and to get user feedback on what needs these applications met (or failed to met), we undertook approximately 150 hours of general observations, user shadowing and interviews with practitioners in five departments of three from the main hospitals in Northern Italy. In this ethnographic work, we could recognize most of the unintended shortcomings of ICT reported by other important works (e.g., (Ash et al., 2004; Campbell et al., 2006)), especially problems related to workflow inclusion in daily practice and paper persistence. The former issue regarded alterations in work dynamics and ergonomic shortcomings in EPR interfaces that we often saw contributing in eliciting bad emotions and frustration in practitioners; the latter issue regarded the observation that practitioners kept using a sort of parallel paper-based record for utilitarian reasons, as original and informal data entry that is compiled before the electronic counterpart and as pocket-size and foldable proxy of the screenshots of their EPR. In particular, some practitioners we interviewed told us that the precise structure of their paper-based forms was often the outcome of a long-lasting stratification of consolidated work practices, agreements and compromises reached between clinicians and administrative staff, and local conventions conceived for a more effective and less time-consuming charting. Allegedly, two advantages of traditional forms over electronic ones are lost with the digitization of paper-based patient records: i) high flexibility and easiness in customizing and modifying the template of official paper-based forms; in fact, these forms were usually prints of electronic documents that were created with regular word processors and that, once intended modifications had been accepted by the hospital management, could be modified just in seconds; ii) high flexibility in document use, i.e., in being free to use whatever document of the record at need without being forced to follow any predefined flow of work: in fact, new forms could be created and adopted in daily practice with no strain and without the need to upset usual practice, as could likely happen, e.g., when new forms have to be filled in either to gather new data for clinical research, to comply with new accreditation standards, or new duties about informed consent.

As first results of this long-term project, we conceived the design-oriented framework, WOAD (Cabitza and Simone, 2009), and developed the first WOAD-compliant application, ProDoc; this is a prototypical documental application that we customized for the hospital domain as a highly flexible EPR (Cab-

Cabitza F. and Zorzato G. (2010).
DEVELOPING A FLEXIBLE ELECTRONIC PATIENT RECORD AS A WEB OF ACTIVE DOCUMENTS.
In *Proceedings of the Third International Conference on Health Informatics*, pages 46-53
DOI: 10.5220/0002721500460053
Copyright © SciTePress

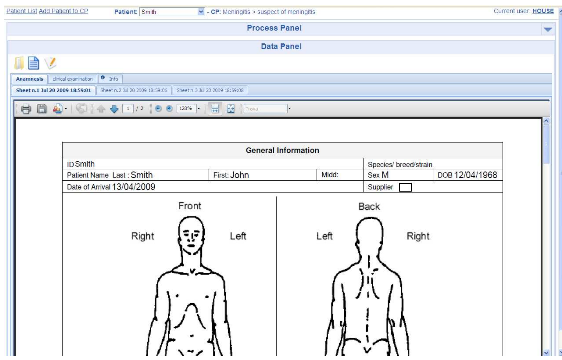

Figure 1: The Panel Data of the ProDoc application system. The prototype presented in (Cabitza et al. 2009b) supported also PDFs to allow for rich visual annotation of documents.

itza et al., 2009b) to address the two requirements above mentioned. In fact, ProDoc allows practitioners to build, customize and use a graphical interface for data entry and retrieval that closely resembles the look and feel of their usual paper-based artifacts so as to mimic the typical interaction with paper forms (see Figure 1). In fact, what in regular EPRs is usually a set of masks to (and views from) the underlying DB, in ProDoc it is a set of persistent documents and forms. Therefore, ProDoc allows users to natively treat and use data in the very terms of the documents they progressively compile. Moreover, ProDoc embeds user-defined active process maps that allows users to get access to any part of the documentation out of any rigid workflow while being aware of the intended flow of activities as it has been defined locally on the basis of practitioners' consensus.

In the next section, we summarize the essential elements of WOAD that underlay the development of ProDoc, i.e., the concept of Active Document, Web of Active Documents and Mechanism. Then, in Section 3 we discuss in more details the WOAD architecture, as it has been deployed in ProDoc, and we illustrate a typical user interaction scenario using ProDoc (see). Conclusions summarize the main advantages of the WOAD framework in the design of EPRs and outline future lines of research.

2 WEBS OF ACTIVE DOCUMENTS

WOAD is a design-oriented framework grounded on the concept of active document (see Figure 2). Each Active Document (AD) can be seen as composed by a "passive" part, i.e., a content container with a specific structure, and an "active" part, i.e., some executable code that provides the passive part with context-aware behaviors. In WOAD, the former part includes the computable definition (i.e., the schema) of modular and scalable data structures, which we call *didgets* (from 'documental widgets'); didgets can be used and reused to build different document templates (where only their topological arrangement changes) that share the same groups of data for different purposes and needs. On the other hand, the active part of an AD is composed by a set of (one or more) executable and modular constructs, which we called *mechanisms*; mechanisms are specialized 'if-then' statements defined over the didgets and their content that can be executed by a WOAD-compliant application (like ProDoc) in order to exhibit document-centered behaviors according to their current data. The seminal idea of an "active documental artifact" was first proposed in (Divitini and Simone, 2000) to refer to data structures capable of assuming an active role in mediating information exchange and coordination among cooperative actors. The most notable research on active documents is the Placeless Document Project developed at PARC (Dourish et al., 2000). Placeless documents are documents that are managed according to their properties, i.e. sort of metadata that both describe the document's content and carry the code to implement elementary functionalities of document management (e.g., automatic backup, logging, transmission). In the WOAD framework, we extend this idea by considering any document and form that practitioners are supposed to fill in and consult in their daily practice as parts of an interconnected document system, i.e., what we call a Web of Active Documents, WoAD. WoADs can be highly customized in different domains and application settings to exhibit active behaviors that support users e.g., in keeping track of

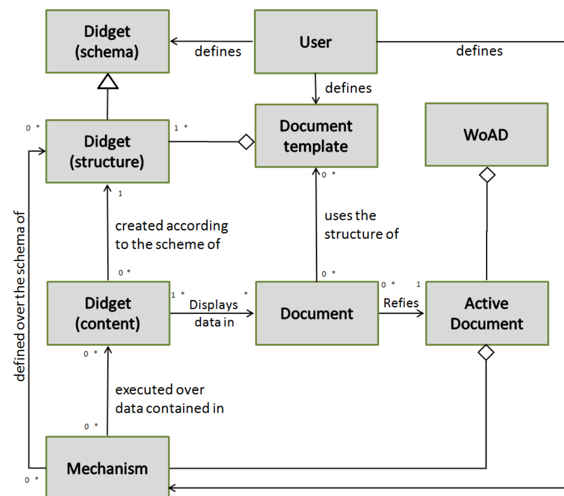

Figure 2: Relationships between WOAD concepts through an UML entity diagram.

the patient's illness trajectory, improving the quality of the information exchanged in shift handoffs and patient handovers, enabling activity- and time-related information retrieval and coordinating collaborative tasks (as e.g., order handling). Accordingly in our case study, we conceived of any single form, record or document used to enter and retrieve clinical data from the patient record (e.g., a therapy prescription sheet) as a single AD that is endowed by local behaviors and is bound to other ADs by means of reactive mechanisms that characterize a specific WoAD. In what follows, we consider in some more details both the passive part of ADs (i.e., didgets and templates in Section 2.1) and their active parts (i.e., mechanisms, in Section 2.2).

2.1 Didgets and Templates

With reference to Figure 3, in WOAD, each document is composed by: i) a set of didgets that are spatially arranged according to a specific document template (see 'Document Template' in Figure 2); ii) sets of data i.e., the document content, that are associated with the didgets contained in the template (see 'Didget Content' in Figure 2).

A didget is defined as a coherent group of form elements, e.g., input fields, iconic elements, buttons, that can be positioned in one or more documents of a WoAD. These elements are gathered together at design time because they relate to either the same abstract data type (e.g., the patient), the same work activity (e.g., drug prescription), or even the same portion of a paper-based artifact, e.g., a table in a record.

Each didget is defined in terms of i) a content model (i.e., data types, constraints, ranges), ii) a layout model (i.e., how it is displayed at the user interface in default of other information) and iii) a set of rendering functions, i.e., executable code that can be interpreted by a client application (e.g., a web browser) to change how the didget's elements and data are displayed. These models build up what we call a didget schema (see Figure 2). Users can place a didget (schema) in any position of a document template and thus create a *structure* of that didget (see Figure 3). A didget structure can be reused in any other document of the same WoAD, so that it constitutes a sort of distributed "entry point" to the same set of data (i.e., the content of the same didget structure). When a user puts a didget structure in a certain position of a document template, she can specify whether the associated group of elements must appear only once in the document (and exactly there) or if users can add more (didget) *content* (see Figure 2) in that structure in tight succession (much like multiple rows in a table) to allow for extemporaneous needs for additional room for data that are not predictable at document design time. For instance, a template of the Anamnesis form can contain a didget to record the examinations previously undertaken by the patient. If the didget has been defined as "multiple", this means that if a physician needs to record more than one examination for a patient into an Anamnesis form document, she can add how many rows (i.e., examinations) she needs and all of them will be stored into the didget. Didgets hold all the data that users feed into them into a permanent memory, time-stamp these data, allow to distinguish between (logically) eliminated, provisional and consolidated data and display these latter data in a last-in-first-out fashion: in this way, users can get access to the whole history of data imputation for any single didget.

Users can create the templates they need by means of the AD Template Editor (ADTE). This is an editor intended to enable users to create document templates through a graphical interface by picking up specific didget schema from a palette of predefined ones and placing them in the draft template in a what-you-see-is-what-you-get manner. Didget schemas can be domain-independent (e.g., regular text-boxes, check-boxes, identification fields) or domain-dependent, e.g., a group of data fields that is related to the identification of patients, or related to drug prescription, like drug name, dosage, administration way and the like. Domain-dependent didget schemas are usually defined by business analysts in cooperation with expert users of the application field and made available to users by a standard palette in the ADTE. The ADTE also provides a second palette containing the didget structures that have already been created in the same WoAD. Users can use this palette in order to reuse the same didgets in different templates. Moreover, users can use the ADTE to define new didget schemas in terms of both the content model (e.g., data fields with their type) and the layout model, i.e., its visual aspect. In addition, the ADTE allows to define the specific rendering procedures of a didget (see c in Figure 3) in terms of Javascript functions, which can affect user interaction with the single didget in each document embedding it by enabling advanced features of text formatting and document rendering. For instance, for a certain didget, users can define a rendering procedure that displays a balloon containing some information regarding a specified field of the didget (the purpose of these procedures will be clear after reading Section 2.2).

Once a template has been defined, users can begin use the documents that are built on that template. In fact, a WOAD document is generated by coupling

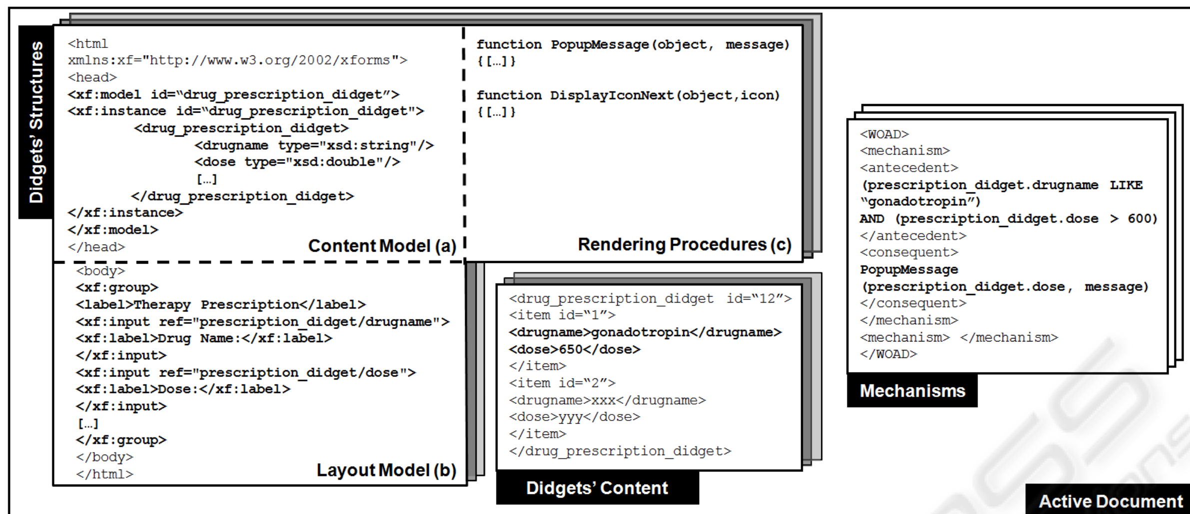

Figure 3: A graphical representation of the components of an Active Document.

the document template, which is associated with a set of didget structures and provides the topological information for their displaying, with the content of its didget structures that is related to the specific document associated to a specific patient. In other words, every different document that is based on the same template (i.e., that is associated with the same didget structures provided by the template) represents an instance of the document template for a specific patient. This means that users can create new documents according to their needs and local practices. For instance, doctors could want to have only one document for each template be associated with a single patients, as in the case of the Anamnesis form, which reports all the past clinical information through multiple didgets that allow for multiple values; conversely, doctors could also want to associate many documents based on the same template with the same patient, as in the case of the clinical diary: this is a case where physicians create a new document every day during the patient's stay to organize their clinical annotations in a time-oriented fashion. In both cases, data are recorded in the "content didget" associated with the didget structures contained in the templates (see Figure 3). In the former case (one document for template with multiple didgets for each patient), doctors can minimize the dispersion of data across different documents and rely on a single "place" to consult; in the latter case (multiple documents for one template for each patient) they can have the didget content be partitioned according to some policy (e.g., the stay's length) and each partition be associated with a different document so as to organize the patient record as they were used to in paper-based folders. In either cases, it is noteworthy that the content of all the documents that are based on the same template refers to the same data structures, i.e., the didget structures that are contained in the template itself: this guarantees WOAD-compliant applications can process data as efficiently as in EPRs based on a traditional DBMS, while allowing a higher flexibility in document use as stated above. Currently, didget schemas are represented using an XForms-like[1] syntax. Once a user has built a didget schema, the ADTE exports it as an XForms form. This contains the XML description of the data model of the didget (see a in Figure 3), and also defines its layout model in XHTML (see b in Figure 3). A document template is an XML description of the specific didget structures that it contains.

2.2 Mechanisms in the WOAD Framework

As said in Section 2, ADs are coupled to sets of WOAD mechanisms that make them "active" and proactive with respect to their content. Mechanisms are rules defined at level of didget structures, i.e., they refer to didgets put in one or more document templates and are triggered according to the didgets content. These if-then constructs can be expressed in any rule-based language (for which an interpreter can be integrated to a WOAD architecture) but we also proposed an abstract denotational language to facilitate their definition by users with little or no experience in programming (see the WOAD language presented in (Cabitza and Simone, 2009)). Mechanisms represent conditions over the didgets' content in their *antecedents* (or *if* parts), and trigger simple actions ex-

[1] http://www.w3.org/TR/xforms/

pressed in their *consequents* (or *then* parts) whenever these conditions are met. Mechanisms' antecedents can contain conditions defined over one or more didgets; in this case, they can refer to didgets that are associated with either only one document template or many document templates of the same WoAD. Mechanisms are triggered by human interaction with documents: any application behavior for which a programming interface is available can be associated to the mechanism's consequent. Accordingly, we classify mechanisms (to rationalize their design) according to what they do on the document content: we then distinguish between mechanisms that (i) modify the content, e.g., to edit or correct values in data fields; (ii) modify content's attributes and metadata, e.g., timestamps, status flags, urgency attributes; (iii) act on the content, e.g., print (parts of) it, check its quality, validate it, consolidate it (e.g., by digital signature); (iv) transmit the content from one system to another through a communication medium, e.g., by email; (v) route documents and build flows of work, e.g., by allowing users link documents of the same WoAD to each other to easily open a document from another, or by allowing users to open/compile certain (portions of) documents only after that also other (portions of) documents have been compiled (and corresponding tasks performed); (vi) change the content's appearance, e.g., by changing the background color or the font style. This last kind of mechanisms act by means of the rendering functions defined at level of didget (see c in Figure 3)and have been object of our recent research on how to improve the ways users access and use document content and on how the content can be displayed to make it more "meaningful". In fact, as researchers active in the field of CSCW, we agree with (Pratt et al., 2004) that EPRs should embed specific functionalities to help practitioners be aware of interdependencies between their work and the activities of others to get an understanding of the collaborative context for their own activity (Dourish and Bellotti, 1992). To this aim, in (Cabitza et al., 2009a) we proposed the concept of Awareness Promoting Information (API), i.e., any annotation, graphical clue, affordance, textual style and indication that could make actors aware of something closely related to the context of reading and writing. The execution of mechanisms can then be seen as a process of API generation, i.e., an operation by which the affordance and appearance of documents and their content is modified, and possibly additional information (e.g., a message) is conveyed to the user in order to make her aware of some condition in the context of document use.

In order to decline the requirements that physicians explicitly expressed in terms of mechanisms, we co-defined with some of their key representatives mechanisms that: could check the correctness of liquid balance values and correct them if necessary (type iii and i); that could mark some values filled in by nurses during a night shift as provisional until the doctor on duty officially double checks them and signs the form (type ii); that could allow practitioners correct a datum without eliminating the previous value for legal concerns (type ii); that could produce official reports printing only parts of the record's content (even distributed across different documents) according to values filled in in specific fields, e.g., the treatment indication, the kind of procedure (type iii); that could prevent users from opening an operation record if the informed consent and the surgery assessment record have not been duly compiled and signed (type v); that could send a copy of an examination request form to the laboratory as soon as all the necessary fields have been filled in and that could send the discharge letter to the family doctor once the hospital stay has been officially closed (both of type iv); and lastly, that would remind practitioners to compile liquid intakes values at regular intervals and that would check that blood examination requests are compiled within noon and raise due alerts if this is not the case so that results can be returned by the end of the day (both of type vi).

3 INTERACTION WITH THE WOAD ARCHITECTURE

In this section, we describe the components of the WOAD architecture (see Figure 4), whose conceptual architecture has been presented in (Cabitza and Simone, 2009), and illustrate how these interact when users are involved in the basic operations of reading and writing an active document, respectively (see Figure 5). We also provide some details about the current implementation of the WOAD components in ProDoc. For our description, we assume that a template has been created through the ADTE and stored in the Template Manager. This is a component that gives both to the ADTE and the main application shared access to templates. When a user asks for a certain document from a specific patient record e.g., a drug prescription form, through the GUI of the application (see step 1 in Figure 5), the request is taken by the Layout Engine. This is a standard component that renders active documents, thus allowing the user to interact with them and communicate with the application. Currently, the Layout Engine can be any regular Internet browser that either supports XForms at client side (like Gecko) or natively supports the output

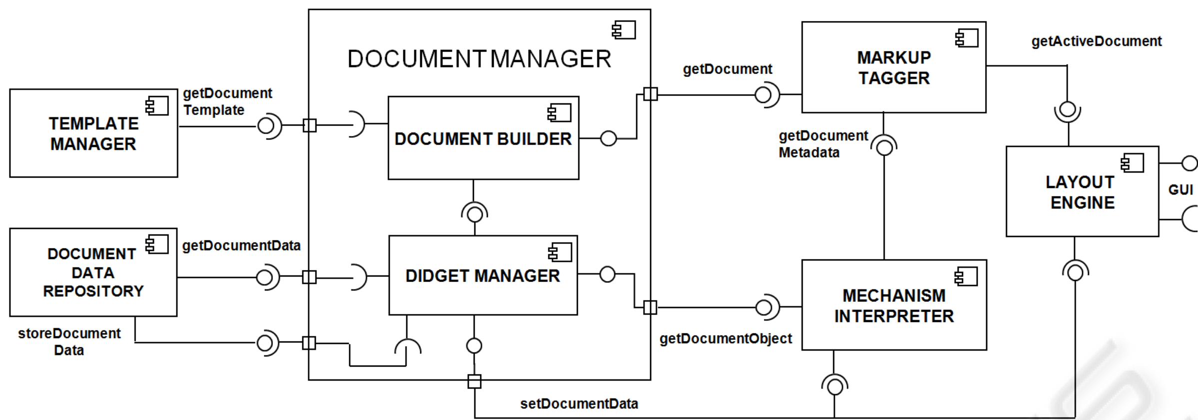

Figure 4: An UML component diagram of the WOAD architecture underlying the ProDoc implementation.

flow produced by an XForms processor at server side (i.e., support the full standards of HTML, CSS and Javascript). The Layout Engine forwards the request to the Document Manager (see 2 in Figure 5) which is the main component of any application based on the WOAD architecture. The Document Manager builds the (passive) document (see Section 2.1) coupling a template with the content related to the document that has been requested by the user in step 1. In addition, the Document Manager provides the data structure on which the application works on the basis of mechanisms (see Section 2.2). To these aims, the Document Manager is divided into two subcomponents: the *Didget Manager* and the *Document Builder*. The Didget Manager creates and keeps the didget structures of a WoAD (see Section 2.1) in its working memory and maintains them synchronized with the *Document Data Repository* (see Figure 4), which is any component that provides data persistence features, e.g., a DBMS. In our current implementation, the Didget Manager is a java class that both loads the content of didgets' structures into correspondent sets of objects, and serializes these objects into the Document Data Repository.

On the other hand, the Document Builder builds an empty form on the basis of a template and fills in it with the (passive) content associated with the specific document requested by the user. The current Document Builder is a java class that creates an Xforms form by joining the schemas of the didget structures contained in a template and associates didgets' content with it (see Figure 2). With reference to Figure 5, the Document Manager retrieves the document template of the requested document from the Template Manager (steps 3a and 4a) and associates it with the correspondent didgets' content provided by the Document Data Repository (steps 3b and 4b) in order to build up the requested document. As soon as the Document Manager gets the document content, it interacts with the Mechanism Interpreter (step 5b) to execute the WOAD mechanisms associated with all the didgets (structures) included in the document. To this aim, the Mechanism Interpreter checks the WOAD mechanisms against the document content provided by the Document Manager, activates the mechanisms whose antecedents are satisfied by the didgets' content and then selects those to execute according to a resolution strategy based on specificity and currentness (Forgy, 1982). The mechanisms' consequents contain instructions that either modify data or build specific metadata to be associated to the document (e.g., metadata that prevent data from being modified or metadata that change the appearance of certain values). This association metadata-document is performed by the Markup Tagger; this component receives both the (passive part of the) document that has been created by the Document Manager (step 5a), and the metadata that has been produced by the Mechanism Interpreter (step 6), and then it translates the metadata either into appropriate rendering attributes (e.g., stylesheet classes) or into calls to rendering procedures (see c in Figure 3). For instance, if the Mechanisms Interpreter has associated a didget text field with the metadata <editable>false</editable>, the Markup Tagger translates it into the HTML attribute "disabled" so that the Layout Engine cannot receive user input for that element. In the current implementation, the Mechanisms Interpreter is based on JBoss Drools[2] and mechanisms are translated into rules that are checked against the didget objects built by the Didget Manager; the Markup Tagger is a java class that embeds javascript code into the Xforms form in order to call the rendering procedures or to modify the style attributes of the XHTML page of the document. The output of the Markup Tagger is then the

[2]http://jboss.org/drools/

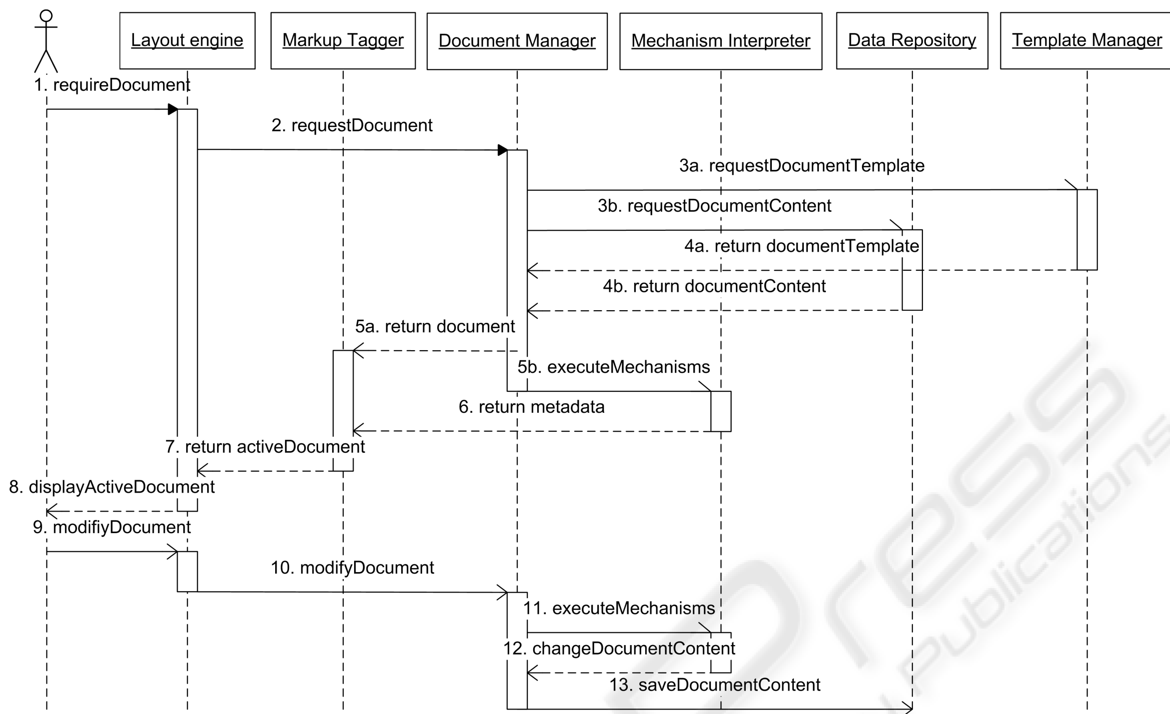

Figure 5: An UML sequence diagram of the typical user interaction with ProDoc.

active document: this presents the requested content with the layout specified by the corresponding template and displays additional information according to the metadata. The active document is sent to the Layout Engine (step 7) which finally displays it to the user (step 8).

On the way round, when a user modifies an active document (step 9 in Figure 5), the Layout Engine sends the single modifications of the content to the Document Manager (step 10). As a consequence, the document (i.e., its didgets) will be automatically re-processed by the Mechanisms Interpreter (step 11) that constantly monitors all the WoAD didgets. In this way, every data change can be immediately captured and processed by the application logic that is formalized in the mechanisms that fit the current context best. To this aim, the Mechanisms Interpreter directly interacts with the Document Manager in order to commit the instructions that update the content (step 12). Finally, the Document Manager stores the documents' updates into the Data Repository for the sake of data persistence (step 13).

4 CONCLUSIONS AND FUTURE WORK

Summarizing, in this paper we have illustrated the concept of Active Document within the WOAD framework: an AD is composed by reusable and modular "field nuggets", called didgets, and it is made active by modular interpretable code, called mechanisms, that trigger behaviors according to the context. This modularity and the accentuated separation of information-related and functional needs (addressed by specific didgets and mechanisms in the passive and active part of a document, respectively) is what makes a *web of interconnected ADs* a suitable electronic document platform that can be reused in different work settings and maintained over time to flexibly address ever changing users' needs. To the present moment, the WOAD framework encompasses (i) a conceptual model and architecture that represent the main entities and relationships involved in collaborative work mediated by complex document systems; (ii) a denotational language by which to express reactive mechanisms in an abstract and platform-independent way (presented in (Cabitza and Simone, 2009); (iii) a set of software components that supports users by means of active documents i.e., electronic documents (in XML format) that exhibit collaboration-oriented behaviors

proactively with respect to the context of work (see Figure 4); iv) a couple of prototypical applications to facilitate users in building their document templates and active mechanisms: namely an active document template editor and a mechanism editor.

The first vertical application system to be based on a WOAD architecture is ProDoc: built as a proof-of-concept and prototypical electronic patient record, it has provided first feedback from key users and preliminary validation from the field of work, as presented in (Cabitza et al., 2009b). In this paper, we have presented the core implementation choices we undertook for the first deployment of ProDoc, based on JBoss Drools (for the mechanism interpreter) and XForms (for the document manager); we also provided some examples of WOAD mechanisms in the hospital domain to give evidence of the advantage that a modular and rule-based approach can give over more traditional approaches that define functionalities at compile-time through an entity-driven requirement analysis and then achieve post-hoc flexibility through mere configuration facilities.

Users stressed the requirement of being supported in defining and maintaining their own data structures and associated behaviors without the heavy involvement of ICT practitioners (mainly to reduce costs and times of intervention). For this reason, our research agenda aims to build one single application that could integrate user-friendly functionalities to build active documents also in a visual and intuitive way. Moreover, we plan to propose a design-oriented methodology to assist IT practitioners in planning and performing digitization programmes of paper-based patient records in a bottom-up and document-centered fashion, in order to minimize the impact of the main unintended shortcomings of ICT programmes in the healthcare domain reported in the specialist literature (Campbell et al., 2006).

REFERENCES

Ash, J. S., Berg, M., and Coiera, E. (2004). Some unintended consequences of information technology in health care: The nature of patient care information system-related errors. *Journal of the American Medical Informatics Association*, 11:104–112.

Cabitza, F. and Simone, C. (2009). WOAD: A framework to enable the end-user development of coordination oriented functionalities. *Journal of Organizational and End User Computing*, 22(1).

Cabitza, F., Simone, C., and Sarini, M. (2009a). Leveraging coordinative conventions to promote collaboration awareness. *Computer Supported Cooperative Work*, 18:301–330.

Cabitza, F., Simone, C., and Zorzato, G. (2009b). ProDoc: an electronic patient record to foster process-oriented practices. In *ECSW'09. Vienna, Austria, September 9-11, 2009*. Springer.

Campbell, E. M., Sittig, D. F., and et al., J. S. A. (2006). Types of unintended consequences related to computerized provider order entry. *Journal of the American Medical Informatics Association 13 Number*, 13(5):547–556.

Divitini, M. and Simone, C. (2000). Supporting different dimensions of adaptability in workflow modeling. *Computer Supported Cooperative Work*, 9(3):365–397.

Dourish, P. and Bellotti, V. (1992). Awareness and coordination in shared workspaces. In *CSCW'92*, pages 107–114, New York, NY, USA. ACM Press.

Dourish, P., Edwards, W. K., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Terry, D. B., and Thornton, J. (2000). Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems*, 18(2):140–170.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19(1):17–37.

Pratt, W., Reddy, M. C., McDonald, D. W., Tarczy-Hornoch, P., and Gennari, J. H. (2004). Incorporating ideas from computer-supported cooperative work. *Journal of Biomedical Informatics*, 37(2):128–137.