WOAD, A Platform to Deploy Flexible EPRs in Full Control of End-Users

Federico Cabitza, Stefano Corna, Iade Gesso and Carla Simone
Università degli Studi di Milano-Bicocca
Viale Sarca 336, 20126 Milano, Italy
+39 02 6448 7815
{cabitza,gesso,simone}@disco.unimib.it

ABSTRACT
In this paper we present the architecture of WOAD, a framework that we propose to make clinical end-users more autonomous in tailoring Electronic Patient Records (EPR) to their changing needs. We provide a short overview of the main concepts of WOAD and then we present the two visual tools that we developed to allow end-users to create their own templates and endow these with proactive and context-aware mechanisms. Finally, we outline the main flows of interaction that have been implemented in ProDoc, a WOAD-compliant prototypical patient record.

Keywords
WOAD, ProDoc, Datoms, Didgets, Electronic Patient Record, End-User, Visual Editor

INTRODUCTION
Electronic Patient Records, under the promises of facilitating accountability and research and improving care efficiency and patient safety, have stimulated important investments in the last twenty years. Therefore this kind of applications are nowadays become more and more common in clinical settings all over the world, although at a diffusion rate that is much lower than initially expected [1]. Recently, an increasing number of scholars propose to ascribe this phenomenon also to the rising perception that going paper-less in hospital wards is but a trivial endeavor [3, 14, 16]. Thus, although it is not an easy task to define what failures in ICT projects really are [4] and although publication bias (i.e. the tendency not to report bad results or failures) certainly affects ICT literature [18, 2] (as indeed many other disciplines), the best estimate is that most EPR projects fail in some way [13]. Analysing the deep reasons why this can happen is out of the scope of this paper. In short, we can summarize these reasons with the fact that traditional EPRs are developed by ICT professionals with scarce or no experience about the clinical domain, and this leads them to not consider the existence of specific local needs (e.g. the needs of a single hospital ward), developing extremely rigid EPR systems. Yet, to present our technological solution we start by taking the stance that designers of EPRs should focus on interaction first and foremost, rather than, e.g. exclusively on data types and data-oriented functionalities. Within the health informatics field, this stance is in line with those who advocate to adopt the interaction design tenets [8] to design information systems that keep the people who will use them "in the loop" and, more yet, give them the control of how the application must be tailored to their specific and local needs.

In this line, in the past years we conducted a number of observational studies (reported in [6]) to elicit the requirements that doctors and nurses perceived as the most important ones to avoid that the incumbent EPR project would end by blowing up in their face, either by requiring them more effort in documental work than the paper-based counterparts, or by imposing organizational constraints and procedural bottlenecks that made sense only on paper. We categorized the main requirements in three classes: *support*, *autonomy* and *flexibility*. In the light of these requirements we conceived an architecture, called WOAD (described in [5]) and realized ProDoc, a prototypical EPR that is based on this architecture, which we described in [7]. While the class 'support' is conceived general enough to encompass all those traditional data-oriented functionalities that support practitioners in carrying out their tasks (e.g., retrieving records by multiple parameter queries, chart printing, calculating liquid balances and other scores), ProDoc was intended as a proof-of-concept application of the latter two: autonomy and flexibility. Obviously these latter are not uncorrelated, indeed we consider flexibility as a necessary requirement to make EPRs "malleable" and tailorable to the ever changing needs of their users; but, differently from the mainstream approaches, we do not think that flexibility can be bestowed on practitioners "from above" but that rather they have to make their EPR flexible "on their own", in an autonomous manner with respect to both the ICT vendor and the ICT specialists. Thus in this paper we intend autonomy as a precondition for actual flexibility and as something that must be guaranteed toward concrete purposes. Specifically, we will present a computational platform that is aimed at making practitioners autonomous in, on the one hand, building and maintaining over time their own electronic documents (seen as modular and reusable components of the EPR user interface); and, on the other hand, in endowing these documents with simple rules that are executed asynchronously by the system according to the context and the content that practitioners progressively fill in.

This platform, WOAD, is then an end-user programming environment where the application layer of the EPR, like

ProDoc is, aggregates patient data in *sets of electronic documents*, as if they were physical sheets of the paper-based record; and where users can employ a specific editor to create the data types they need to document their work almost "on the go" and place the corresponding fields and input elements in the templates of these sheets, in a manner that purposely mimic the way they used to edit the templates of their paper-based charts with a regular word processor. In addition to the template editor, WOAD also encompasses a visual rule editor, by which practitioners are facilitated in creating small bunches of if-then logic; these simple rules are intended to be local to the clinical documents and to be defined according to the conventions currently in use in a specific department [6]. To this aim, the editor allows the practitioners to create specific conditions or data patterns over the templates they have previously created. Although rules can act on any part of a document, we advocated their creation especially to modulate how the document's content looks like, and therefore to convey what in [6] is called Awareness Promoting Information (API), i.e., any additional indication that could help practitioners become aware of what is going on in their setting [10] and recall knowledgeable ways to cope with the situation.

In the following sections, we will briefly outline the WOAD architecture and see how its components, including the template and rule editors, interact with the end-user to present clinical data with the typical flexibility provided by the still efficient and versatile paper-based patient record [11].

THE WOAD FRAMEWORK

WOAD is a *design-oriented* framework that encompasses both a *conceptual model* and a *reference software architecture*, and is grounded on the concepts of "active document" and "web of documental artifacts" [5]. In WOAD, documents are composed by two intertwined parts: a *passive part* and an *active part*.

The passive part contains the content users fill in and arranges it according to a *template*. This defines how *didgets* ('documental widget') are topologically arranged. A didget represents the reusable instance of a *datom* ('documental atom') within a specific document. Datoms are modular data structures encompassing a set of data fields that coherently represent a specific aspect of the reality of interest. The active part is composed by a set of mechanisms, i.e., specialized 'if-then' statements that augment the passive part of a document with context-aware and proactive behaviors. A mechanism can be defined over either datoms, didgets or their content and is triggered according to the current contents of the document.

DOCUMENT TEMPLATE EDITING

The Active Document Designer (ADD) represents the means that allows the end-users to create the WOAD document templates, and consequently the datoms, that they need. To this aim, the ADD (Figure 1) encompasses two distinct visual editors: the Datom Editor (DE) and the Template Editor (TE). While the DE allows the users to create the datoms by defining both the data model (e.g., the data type of a field) and the layout model (e.g., the visual aspect), the TE allows for the graphical design of the topological arrangement of the documents.

A user who wants either to create or edit a document template has to pick up the datoms from a palette (or stencil), which contains all the datoms that have been previously created with the DE, and place them into the drawing area that represents the document. Once a datom has been placed in the template, the corresponding didget is created and added to another palette, which makes available the didget outside of the document in which it was created and allows for the reuse of the same didget into other document templates.

The reuse of the didgets allows for sharing data between different documents, either if those documents are based on the same template or on a different one. Moreover, the didgets can be also used for sharing data regarding different resources (e.g., all the patients of a hospital ward). This feature of the didgets is specified through their `global` attribute, which can assume four different values (see Table 1): G0) the didget holds some pieces of data that are local to a specific instance of a document (*local data*), e.g., the value of the daily measurement of the patient's temperature that practitioners inscribe on the *Daily Sheet* (DS); G1) the content of a didget is shared between all the instances of a document based on a particular template and related to a specific resource; G2) the didget shares some pieces of data between the instances of some documents that are based on different templates, but that are related to a single resource, e.g., some portions of a patient's personal data (like the patient ID, her name and surname); G3) the contents of a didget are shared between all the document instances without any constraint both on the template and the resource. When a user who is editing a template drops a datom and creates the related new didget, the latter will hold only local data by default (i.e., the `global` attribute is set to the G0 value). The user can set the desired level of globality of a didget simply using a graphic menu that appears directly under the graphic representation of the didget in the template draw area.

In a similar way, through the same graphic menu, the users can also specify if a didget must display its set of fields only once ("single didget") or if this set of fields has to be repeated in tight succession ("multiple didget") for a certain number of times. For instance, the latter case is useful to handle the need to organize some data in tabular format that could be repeatable on the same instance of a document (e.g., the vital parameters of a newborn within few moments from delivery), using the structure of the didget (i.e., the related datom) to define the organization of the rows.

When a user has finished the composition of the document template, she stores it into the *Template Manager* (see below for details about this component). If the user has opened an already existing document template in order to modify it, the storing operations are not destructive and adopt a simple versioning system: each version of a template is labelled with the timestamp of its creation.

Making the users able to build their documents in a what-

	Data Shared Between		
	Instances	Templates	Resources
G0	✗	✗	✗
G1	✓	✗	✗
G2	✓	✓	✗
G3	✓	✓	✓

Table 1. The levels of the `global` attribute of a didget.

you-see-is-what-you-get manner allows for increasing timeliness, high flexibility and 'tailorability' with respect to both creating and modifying operations.
An exemplificatory scenario is the need to add an element to a document (e.g., a checkbox) by which clinicians can indicate whether the patient has expressed the informed consent or not. Here the constraint associated with this document element is that all fill-in operations on any part of the document must be inhibited if the informed consent checkbox has not been marked.
Usually, in a traditional information system, addressing this need would require to apply a set of changes that may potentially involve the whole system and that have to be necessarily call for the involvement of software analysts and developers. Adopting this approach requires the users to wait until the software professionals have completed all the necessary tasks to make the required changes to the system.
On the other hand, using the ADD, the users can directly and quickly add any new feature to their documents, without involving any other professionals. They have just to edit the document template that has to be modified, opening it with the TE. Once the template has been opened, the user has just to select the previously created (with the DE) "Informed Consent" datom, dragging it over the template, and drop it at the desired place. In a similar way, also the application logic that prevents from or enables the editing of the fields in the same document could be added using the *Mechanism Editor* (see the next section for more details).

Currently, the TE is a prototypical application based on the *Oryx Editor*[1] (see [9]). The Oryx Editor is a web-based, extensible editor, that has been initially conceived to model business processes. This editor adopts a plug-in architecture that facilitates its extension by adding other graphic editing features (e.g. the set of third party plug-ins to model the XForms, Workflows or the Petri Nets).
The Oryx Editor is also coupled with another web application, the *Oryx Repository*, which acts as a simple "file manager" and allows for storing, browsing and managing the various models that the users create through the Oryx Editor. Due to its simplicity, we used the Oryx Repository as the user frontend of the Template Manager.

MECHANISM EDITING

The Mechanism Editor (ME) is the tool that allows the end-users to create and edit their mechanisms. ME is a web editor based on Oryx Editor, as well as the TE and the DE. ME provides users with a simple GUI that allows to create the mechanisms. The composition of the mechanisms is mainly based on drag & drop, in order to facilitate the use of the ME for those users with little or no experience in programming. The GUI is split into three areas (see Figure 2). The top section of the left area contains the list of all existing templates (previously created with ADD). A template can be dropped both to the central and the right areas, in order to respectively build the if-part and the then-part of the mechanism. The bottom section of left area contains the list of all the saved mechanisms. A user can load an existing mechanism by performing a double click on the mechanism item in this list. The left area also contains the trash area (like in modern desktop environments) in which users can drop any action or condition that they want to delete.

The if-part of a mechanism contains the conditions that the system must match to the document content; those conditions are defined on one or more didgets that are contained in one or more templates, as well as on basis of environmental variables, like system time and current users. A condition is composed through an interface that appears when a template has been dropped into the central area. This interface is unique for each of the dropped templates, and is composed by a form and a table where all the created conditions are listed. The users compose their conditions through the form area that contains three dropboxes and a textfield. The dropboxes allow respectively for the selection of the didgets, the fields and the constraints. The constraints are filtered according to the selected field data type (e.g., for a numeric field, the constraints will be "greater", "lower", "equals" and "not equals"). The textfield is used to complete the condition with the value of the constraint. The then-part contains the actions to be triggered when the above mentioned conditions are met. An action can be composed in the same way of the conditions, but in this case the third dropbox contains the list of the available API (e.g., 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 the mechanisms can be seen as the process of API generation, i.e., some operations by which the affordance and the appearance of the documents and their content are modified, and additional information (e.g., some messages), if any, are conveyed to the user. Each type of API has unique parameters. When a new action is added, the ME shows a property window that contains a form with the API parameters (e.g., the *Criticality API* changes the field color, and consequently the related property window contains a color palette).
The actions can be defined to act both on the same document in which conditions are met (e.g., an action modifies the color of the temperature field when its value is higher than 40 degree) and on some other documents (e.g., an action creates an alert message in all the documents if the patient suffers drugs allergy).

In our scenario, the user needs to create a mechanism that inhibits the fill-in operations on the document where the Informed Consent checkbox is placed, if this has not been checked. The user starts the composition picking up the document template from the list in the left area and dropping it

[1]*Oryx* is an academic project that is mainly developed by the *Business Process Technology* research group at the *Hasso-Plattner-Institute* (`http://bpt.hpi.uni-potsdam.de/Oryx/`).

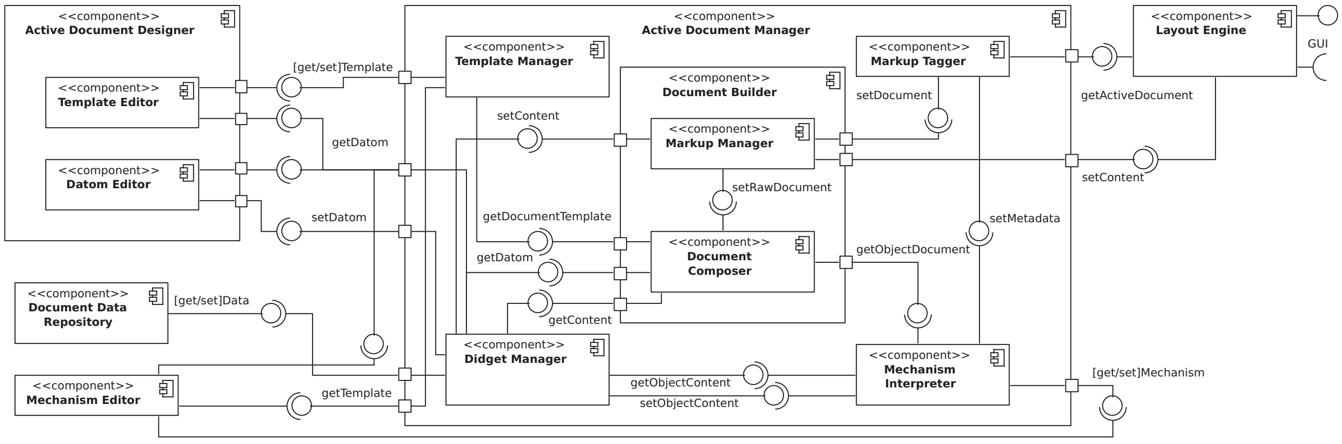
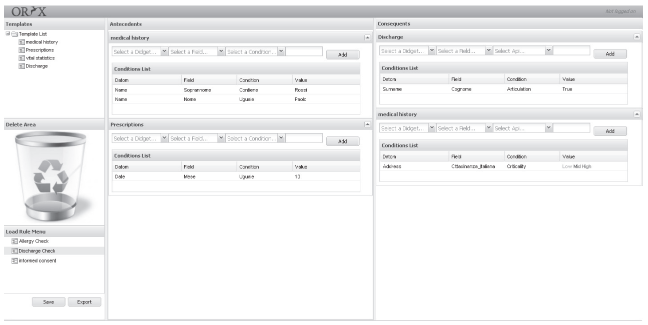


Figure 1. The UML diagram of the components of the WOAD framework.



medical history

Conditions List

Datom	Field	Condition	Value
Name	Soprannome	Contiene	Rossi
Name	Nome	Uguale	Paolo

Prescriptions

Conditions List

Datom	Field	Condition	Value
Date	Mese	Uguale	10

Discharge

Conditions List

Datom	Field	Condition	Value
Surname	Cognome	Articulation	True

medical history

Conditions List

Datom	Field	Condition	Value
Address	Cittadinanza_Italiana	Criticality	Low Mid High

Figure 2. The Mechanism Editor user interface.

into the central area. Then, she selects the informed consent field from the fields dropbox, and "equals" from the constraints. Finally, she writes in the textbox the "false" value and pushes the "Add Condition" button. Once the if-part has been created, the user starts to compose the then-part. The user drops the previously chosen template in the right area, and selects all the fields that she needs to protect. Then she adds the action that makes a field read-only, and pushes the "Add Action button" to complete. Once mechanism is defined, the user saves it into the local repository (for future modifications), and then she converts the mechanism into the rule engine dialect (i.e., the Drools DRL) in order to make it available for the *Mechanism Interpreter* (MI).

INTERACTIONS BETWEEN WOAD COMPONENTS

With respect to the architecture depicted in Figure 1, in this section we describe how WOAD components interact when a user requires to read and update an active document (see also the steps in Figure 3); at the same time, we will provide a short description of these components and some details about their current implementation in ProDoc. This description is based on the assumption that the document templates have already been created with the ADD and stored into the TM. Similarly, we describe the process of mechanism creation.

When a user asks for a document, the GUI of the application sends the request to the *Layout Engine* (step 1 in Figure 3). The LE allows for displaying and interacting with the documents, and currently it is any regular web browser that is fully compliant with the W3C standards (i.e., HTML, CSS and JavaScript). The request is forwarded (step 2) to the *Active Document Manager* (ADM), the main component of the WOAD architecture, which builds the passive part of the document, and provides the data structures to support the execution of the mechanisms. Due to its complexity, the ADM is composed by five subcomponents: the above mentioned *Template Manager* (TM), the *Didget Manager* (DM), the *Document Builder* (DB), the *Mechanism Interpreter* (MI) and the *Markup Tagger* (MT).

The TM manages the templates and the related versioning capabilities, and provides the access to the templates to the other components of the framework. The DM creates and manages the didgets, provides the other components with the access to the definition of the datoms, and works in conjunction with the *Document Data Repository* (DDR), a component that provides data persistence features[2], to keep the didgets synchronized with their contents.

The DB builds an empty document and, if needed, fills in it with the related contents of the didgets. To accomplish this complex task, also the DB is divided into two subcomponents: the *Document Composer* (DC) and the *Markup Manager* (MM). The DC composes the document (steps from 3 to 12) by coupling the topological arrangement, the datom definitions (both UIs and data models) and the contents of the specific document instance, if any, and produces the representation of the whole document using an intermediate format (i.e., XHTML and XForms). The DC retrieves the instance of the document from its internal memory, and queries the TM and the DM respectively for the template, the datoms and the contents. In this phase, the MM acts as a translator[3], getting the intermediate representation of the document from the DC and transforming it into a markup language expression (i.e., HTML) that the LE is able to render (steps 12 and 13), with no additional tool. Once the MM has finished with

[2]The DDR is a Java package based on the *HyperJAXB* library (`http://java.net/projects/hyperjaxb3/`).

[3]The MM is a customization of the *betterFORM* XForms processor (`http://www.betterform.de/`).

the intermediate document, it sends the passive part of the document to the MT (step 13). The MT[4] forwards the received document to the LE, and this latter displays the document to the user (steps 14 and 15).
On the way round, if the user makes some changes on the document content (step 16), the LE forwards them to the MM (step 17). Consequently, the MM invokes the DM (steps 18 to 21) to update its internal data structures. Finally, the DM sends the new contents to the DDR (step 19) for the sake of data persistence.

Asynchronously with respect to the other operations, the MI[5] constantly monitors both the data structures that the DC and the DM maintain in their working memory and the execution context. When the MI detects that something has changed (e.g., the user edits some document), it checks the mechanisms, activating and executing those in which the if-part is satisfied, following a resolution strategy that is based on specificity and currentness [12]. When a mechanism is activated, the MI executes the operations defined in the *then-part* (`alt` in Figure 3) that can either modify the contents (step 25) or generate some *metadata* (step 22) to alter either the document aspect (e.g., changing the appearance of a field) or its behavior (e.g., making a field not writable).
When the MT receives the metadata, it translates them into rendering attributes (e.g., CSS classes) and procedures (e.g., JavaScript functions), and sends the latter to the LE (step 23). Finally, the LE updates the active document, displaying the new styles and running the new procedures (step 21).

The users can create their own mechanisms (step 1 in Figure 4) using the Mechanism Editor (ME). The ME is a standard component that allows the users to create, edit and export mechanisms. ME requests to the TM the list of available templates (step 2). The TM returns the current template list (step 3), and the ME fills its template menu and creates an empty mechanism (step 4).
The user starts to compose a mechanism (step 5) picking up a template in the list and droping it in either the if-part or the then-part areas. When a template has been dropped, the ME requests to the TM the list of the datoms that have been used in this template (steps 6 and 7). Once the ME has received this response, it fills the list of datoms, and for each of these datoms the ME requests the list of their fields to the DM (steps 8 and 9).
Since the ME gets all these data, the user can start to set the conditions and actions that define the mechanism (steps 10 and 11). Once the composition of the mechanism has been completed (step 12), and the user has requested to save it, the ME processes its internal data structures (i.e., the lists of conditions and actions) and stores them into a persistent storage medium (e.g., a file). When the mechanism has been saved, the user can convert it, and this process consists in the translation of the mechanism into the Drools language, and makes the result available to the MI.

[4]The MT is a Java class that makes active the document coupling its passive part with the metadata produced by the MI.

[5]The MI is based on JBoss Drools (`http://jboss.org/drools/`)

CONCLUSIONS AND FUTURE WORK

The paper illustrated two main functionalities of the WOAD architecture: the first one allows for the flexible and modular definition of the structure of the electronic charts, forms and documents that mediate care and collaboration in hospital settings. The second functionality allows practitioners to define simple rules and associate them to the documents so that their content can be visually and textually enriched (e.g., in terms of different affordances) according to the context and at various levels of scope, from the hospital-wide level to even the single practitioner one. In this paper we described these functionalities in terms of both the user interface that supports them and the architecture that realizes their implementation. Both the prototypes have been tested and used in a laboratory environment for one month by volunteer students that were called to digitize the charts and forms used in two real hospital settings. This user session helped the development team detect and correct the main anomalies and identify improvement areas in the experience of unskilled practitioners.

As said in the Introduction, these two features are intended to enrich a prototypical application we deployed in the hospital domain as an innovative and yet lightweight EPR, ProDoc. Yet, this application can be also seen as a demonstrator of a wider class of applications supporting collaborative work. In WOAD compliant applications, coordination is achieved mainly through documents, with respect to both their visible structure and to that particular kind of additional information that can be conveyed through the user interface to promote "collaboration awareness" [10]. In particular, this information is conveyed according to simple rules that end-users are called to visually create even if they have no specific IT skill, let alone programming skills. This is the most challenging part of our research program, which places it within the scope of both the End-User Development and Interaction Design fields.

Consequently, our future work will focus on how to present and afford these functionalities for different classes of users in order to modulate this kind of support according to their technical skill and domain expertise. To this aim, the empirical work that inspired the conception of the WOAD framework and its proof-of-concept application, ProDoc, will continue to both improve its "malleability" [17] to the work context and to validate its applicability in other domains where we have gained an initial positive feedback [15].

REFERENCES

1. D. Balfour, S. Evans, J. Januska, H. Lee, S. Lewis, S. Nolan, M. Noga, C. Stemple, and K. Thapar. Health information technology - results from a roundtable discussion. *J. Manag. Care Pharm.*, 15(1):10–7, 2009.
2. P. A. Bath. Health informatics: current issues and challenges. *J. Inform. Sci.*, 34(4):501–518, 2008.
3. M. Berg. Implementing information systems in health care organizations: Myths and challenges. *Int. J. Med. Informatics*, 64(2-3):143–156, 2001.
4. C. Sauer. *Why information systems fail: a case study approach*. Alfred Waller Ltd., Publishers, 1993.
5. F. Cabitza and I. Gesso. Web of Active Documents: An Architecture for Flexible Electronic Patient Records. In A. Fred, J. Filipe, and

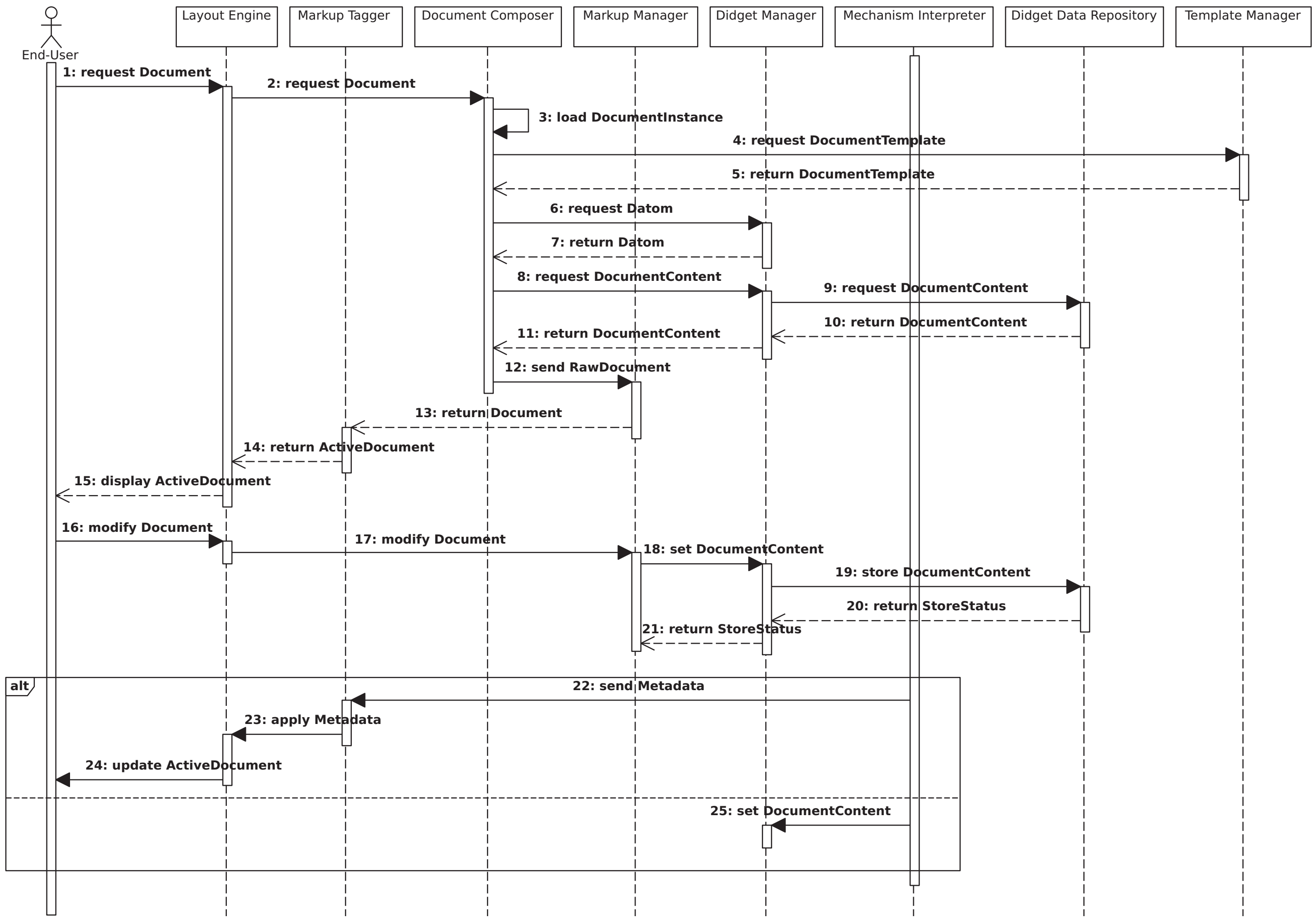


Figure 3. The UML sequence diagram of the typical interactions between the components of the WOAD framework.

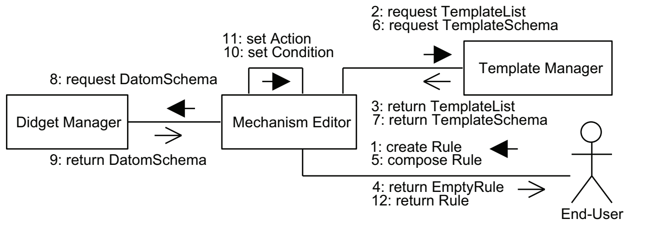


Figure 4. The UML collaboration diagram of the rule creation process.

H. Gamboa, editors, *Biomedical Engineering Systems and Technologies*, volume 127 of *CCIS*, pages 44–56. Springer Berlin Heidelberg, 2011.

6. F. Cabitza, C. Simone, and M. Sarini. Leveraging coordinative conventions to promote collaboration awareness. *CSCW*, 18:301–330, 2009.
7. F. Cabitza, C. Simone, and G. Zorzato. ProDoc: an electronic patient record to foster process-oriented practices. In *ECSCW'09*, pages 119–138. Springer, 2009.
8. E. Coiera. Interaction design theory. *Int. J. Med. Informatics*, 69:205–222, 2003.
9. G. Decker, H. Overdick, and M. Weske. Oryx – An Open Modeling Platform for the BPM Community. In M. Dumas, M. Reichert, and M.-C. Shan, editors, *Business Process Management*, volume 5240 of *Lect. Notes Comput. Sci.*, pages 382–385. Springer Berlin/Heidelberg, 2008.
10. P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *CSCW'92*, pages 107–114, New York, NY, USA, 1992. ACM Press.
11. G. Fitzpatrick. Understanding the Paper Health Record in Practice: Implications for EHRs. In *HIC2000*, Adelaide, Australia, 2000.
12. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intell.*, 19(1):17–37, 1982.
13. R. Heeks. Health information systems: Failure, success and improvisation. *Int. J. Med. Informatics*, 75:125–137, 2006.
14. M. Jones. "Computers can land people on Mars, why can't they get them to work in a hospital?" Implementation of an Electronic Patient Record system in a UK Hospital. *Meth. Inform. Med.*, 42:410–415, 2003.
15. M. Locatelli, A. Viviana, and F. Cabitza. Supporting learning by doing in archaeology with active process maps. In *eLearning 2010*, 2010.
16. C. Nowinski, S. Becker, K. Reynolds, J. Beaumont, C. Caprinia, E. Hahn, A. Peresa, and B. Arnold. The impact of converting to an electronic health record on organizational culture and quality improvement. *Int. J. Med. Informatics*, 76(1):S174–S183, 2007.
17. K. Schmidt and C. Simone. Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. *CSCW*, 5(2/3):155–200, 1996.
18. W. Tierney and C. McDonald. Testing informatics innovations: the value of negative trials. *J. Am. Med. Inform. Assoc.*, 3(5):358–359, 1996.