XoWiki Content Flow – From a Wiki to a Simple Workflow System*

Gustaf Neumann

Institute of Information Systems and New Media,
Vienna University of Economics and Business Administration
Augasse 2-6, A-1090 Vienna, Austria
gustaf.nemann@wu-wien.ac.at

Abstract. This paper introduces XoWiki Content Flow, which is a simple workflow component based on a state transition system. This workflow component is used as an extension of the XoWiki framework [1, 2], a wiki based environment for content management applications. The primary application focus is on managing state and transitions of content together with an application specific set of attributes. A user can define with the system different kind of application objects (such as shared online documents, or multiple choice questions and answers) that behave differently depending on their state. Technically, the content flow package is defined as a sub-package of XoWiki and inherits all its functionality of XoWiki. The paper focuses on the the basic principles and design criteria of the package and will present a simple application example.

1 Introduction

During the last years various XOTcl [3, 4] based components were developed for the OpenACS Framework. The flagship components are xotcl-core (the basic functionality) and XoWiki (a wiki based content management system developed with a Web 2.0 mindset; see e.g. [1, 2]. While XoWiki started out as a basically a wiki system, it developed during the last year into a flexible content management framework [5], where many of its agile concepts can be reused in other contexts as well. Examples are e.g. the s5 package (available via the public CVS system of OpenACS) or iLogue and Mupple [6].

This paper introduces a workflow component for the XoWiki framework, which extends the functionality of plain XoWiki. Current XoWiki versions allow already to define application specific classes with arbitrary meta-data via XoWiki forms. XoWiki allows to create instances for these application specific classes via web. For example, one can create a survey as an application specific class, where the instance data of the survey entered by the participants is seen as instances (entries) of the form. Similarly, one can create multiple choice questions, or shared documents with more meta-data etc. via the same interface. The XoWiki Content Flow package extends this functionality by adding

* Published in: "*Proceedings of 7th OpenACS / DotLRN Conference*", Valencia, Spain, Nov. 18-19 2008.

explicit state and transition management to it. The implemented package is a sub-package of XoWiki and implements workflows via mixin classes [7, 8] for the XoWiki classes. In the following sections we will first focus on the basic principles and design criteria. Then we will present a simple application example and finish with conclusions and related work.

2 Workflows and State Transition Systems

Modeling of process aware information systems has a long tradition especially in the area of workflow systems. When designing a general purpose workflow engine the first question concerns typically its formal foundation. Most of the workflow literature is based on Petri nets (also called place transition networks) which are a mathematical modeling language for describing discrete concurrent systems with non-deterministic execution (in cases, where multiple transitions can fire). The Petri net formalism is frequently used to provide a formal basis for higher level workflow semantics, as for example for the workflow patterns [9].

Another formalism for providing exact process semantics are algebraic methods, such as the process calculus (also called π calculus [10]), which is a successor of the calculus of communicating systems [11]. A central aspect of these approaches is to model parallel behavior by modeling the communication between the participants. These algebraic specification languages are lately often presented as an alternative to Petri nets and vice versa (see e.g. [12]). Formal models like Petri nets and π calculus are required to provide formal semantics and analysis methods. However, as van der Aalst argues, there are unfortunately only a few cases where the formal models are used to actually improve the quality and applicability of the workflow languages [12].

We are in this paper less interested in process semantics (how different processes are synchronized), but more on defining state specific behavior of application objects and the modeling of actions leading to transitions. The presented approach is best described by an formalism developed for operational semantics of dynamic systems, namely on *labeled state-transition systems* (see e.g. [13]). Informally, a labeled state-transition system is based on a set of states S with labeled transitions. The labels on the transitions represent actions. Labeled state-transition systems are defined by a ternary relation of the form

$$p \xrightarrow{\alpha} q$$

where p and q are elements of S and α is from a set of labels Λ. Labeled state-transition systems differ from finite state automata by allowing an infinite set of states S and an infinite set of labeled transitions Λ.

Labeled state-transition systems can be used to model state changes in an information system explicitly. So, the focus of this paper is more on modeling *state aware information systems* rather than *process aware information systems*, but there is certainly a duality between these approaches. The term state aware information systems actually means that the systems handles application specific state changes in an explicit manner.

3 State Aware Objects and State Traces

In our approach the state awareness is realized via state aware application specific classes which model states and actions leading to transitions explicitly. We call these classes shortly *state aware classes* and the instances of these classes *state aware objects*.

For every state aware class we define a finite set of state objects and a finite set of actions (causing transitions). When a state aware class is instantiated, a state aware object is created in some initial state. The full state of the state aware object is defined by the state object and additionally by an application specific set of attributes A. These attributes might be modified and extended by the actions α. Every action can be programmed with application specific semantics and can behave differently depending on the state objects and the attribute set A. When some action of a state aware object are executed (e.g. via user input or an external event) a new state can be reached. For every reachable state a sequence of transitions starting from the initial state can be derived. We refer to such a sequence of transitions as a *state-trace*. Every state of the state-trace consists essentially of a state object s_t and the set of application specific attributes A_n where the subscript denotes n-th transition.

$$\begin{pmatrix} a_{1,0} \\ a_{2,0} \\ \dots \\ s_0 \end{pmatrix} \xrightarrow{\alpha_1} \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ \dots \\ s_1 \end{pmatrix} \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} \begin{pmatrix} a_{1,n} \\ a_{2,n} \\ \dots \\ s_n \end{pmatrix}$$

Note that the sequence of transitions might contain iterations (loops) and that it is possible that e.g. two subsequent execution states might use the same state object s. Every state-trace represents a concrete flow of actions or the concrete control flow of a *workflow instance*. In the developed system all state-traces are stored persistently in the database to make it possible (a) to introspect at any point in time the sequence of transitions leading to the current state, (b) to change the change the workflow to go back to an earlier state and continue from there and (c) to perform mining techniques to analyze traces ex-post and to visualize e.g. collaboration graphs [1].

4 State Specific Behavior

The State Design Pattern [14] is a behavioral design pattern which was developed to implement state specific behavior of objects. The State Pattern allows to extend and influence the behavior of an object in a state specific way without forcing a developer to add case statements to all methods which should behave differently depending on the state. Therefore new states can be introduced in an easy maintainable way.

Figure 1 shows the basic class structure of the State Pattern as suggested by [14]. The state is embedded in some context and different concrete states use the same interface as the abstract state. Many variants of this design pattern

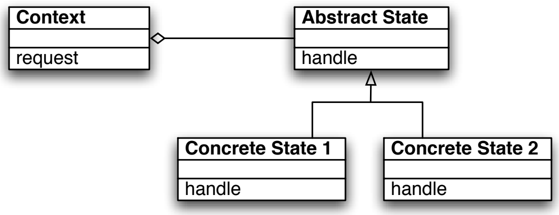

Fig. 1. Structure of the State Design Pattern [14]

were suggested over the last years, such it is today appropriate to speak about a family of state patterns.

For the XoWiki Content Flow, we need three aspects to be handled which are not part of the original State Design Pattern:

1. Since the applicable *actions* of state aware objects are state dependent these should be properties of the state objects. This is needed to implement the flow of labeled state-transition system.
2. The actions are either triggered by the user via *forms* or via external events. The applicable forms are as well state dependent and should be as well properties of the state. These forms provide the user interface to state aware objects with their application attributes.
3. Since the properties *actions* and *form* might be different these must be stored for every state explicitly. Therefore it is more straightforward to implement states as state object and not as classes (as in Figure 1). Note that we have already referred to state objects in the previous section.

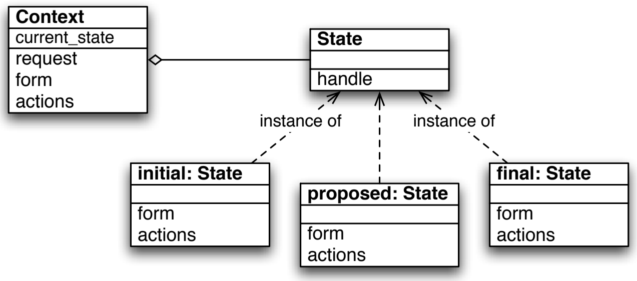

Fig. 2. Realization of State-Pattern with State Objects and Object Specific Methods

To implement the state objects, XOTcl's facility of providing object specific methods is very useful. So, instead of using classes for the concrete states as in class structure in Figure 1, an XOTcl implementation can use state objects without loosing expressiveness. Figure 2 shows an example with three state objects named *initial*, *proposed* and *final*. Typically when instances of the class *Context* try to determines the applicable user interface *form* or *actions*, they delegate this invocation to the current state. Note that e.g. the state property *form* can be implemented in XOTcl either as an attribute or as a method with the same interface.

5 Workflow Definitions

So far we have described how state aware objects are processed, but not, how the application specific state changes are modeled. Figure 3 shows a simple motivating example describing the TIP process of the OpenACS community. A TIP (abbreviation of a technical improvement proposal) is initially proposed, later it might be accepted or rejected, and finally, if accepted, it might be implemented. At any time a TIP document might be edited by a user with sufficient rights.

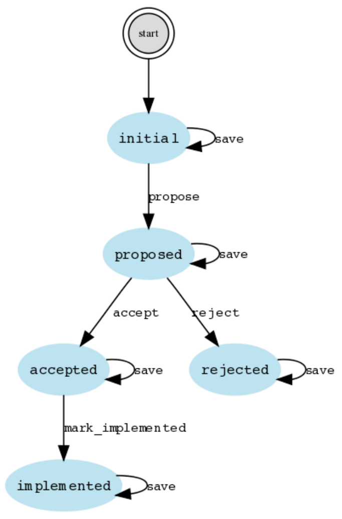

Fig. 3. State Graph for the TIP Workflow

The XoWiki Content Flow system uses essentially three kinds of constructs to define a workflow. These are the classes *State*, *Action* and *Condition* as shown in Figure 4. An actual workflow definition consists essentially of named XOTcl objects for these classes. The *State* objects contain primarily the information about applicable actions and user interface definitions (forms). *Action* objects know their next state and can contain a method *activate* for application specific program code. Named *Conditions* are primarily for conditional control flow branches (like guard conditions in UML's activity diagrams). Condition objects can be be displayed in the workflow graph. In addition to these basic functionalities, XoWiki Content Flow provides means for expressing context-specific behavior. So it is e.g. possible to express in the workflow definition that e.g. in a certain state an administrator (a user with administration rights) is provided with a different user interface (with a different form) than an ordinary user. Similarly, it is possible to define which actions should be offered to users in which kind of roles. Figure 4 shows the actual class structure for work flow definitions as used in the XoWiki Content Flow package.

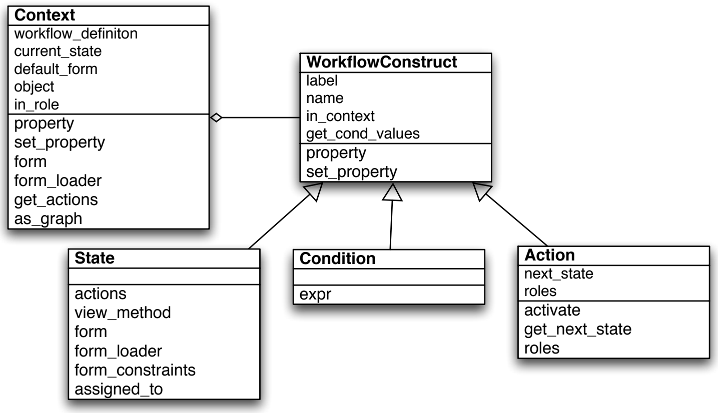

Fig. 4. Basic Workflow Classes

In order to define a workflow for the TIP example (Figure 3) we define the state objects `initial`, `proposed`, `accepted`, `rejected` and `implemented` as well as objects for actions named `propose`, `accept`, `reject`, `mark_implemented` and `save`. The definition of the default form is used for all states, unless a state provides its own form. Forms are as well named objects and are stored in the OpenACS content repository as objects of the type ::xowiki::Form.

```
set default_form "en:tip-form"

Action save -roles admin
Action propose -next_state proposed
Action accept -next_state accepted
Action reject -next_state rejected
Action mark_implemented -next_state implemented

State initial  -actions {save propose}
State proposed -actions {save accept reject}
State accepted -actions {save mark_implemented}
State rejected -actions {save}
State implemented -actions {save}
```

In the current implementation, the workflow definition is entered via the XoWiki form interface. For this purpose, a form field type of the class *workflow* is provided. When a workflow definition is edited, a text editor is opened, when the definition is viewed, the text is transformed into a state graph (see e.g. Figure 3). By using XoWiki forms, the workflow definitions are as well stored with revisions in the OpenACS content repository.

When a workflow definition is provided, it can be used to create a workflow instance of it. This workflow instance is a state aware object as defined in Section 3. State aware objects can be typically *viewed*, *edited* or be *deleted* by users having sufficient rights. When a state aware object is edited the actions defined by the workflow definition for the current state (fulfilling the context constraints) are presented to the user as HTML FORM buttons. When an action button is pressed the actual form data is used for an update and a transition to a potentially new state is recorded. In general, actions can not only be activated via form buttons, but as well programmatically by remoting calls (via method `call_action`) or via scheduled calls at a certain time (via method `schedule_action`).

Since every action activation leads to a new revision in the database, the state-trace is recorded automatically by the underlying framework. The state-trace can be introspected by looking at the revision history of the workflow instance. By making an old revision current, it is possible to jump back to earlier states of the workflow. Note that the workflow definitions, the user interface definitions (the default form in the example above) and the workflow instances are stored in the OpenACS content repository. All these objects are revisioned with exactly the same mechanisms. Since all these objects are actually XoWiki objects, the methods provided by the XoWiki framework can be used these objects as well (e.g. export, tagging, categories, collaboration graphs [15], etc.)

6 Related work

We concentrate in this section on the related work in OpenACS. The OpenACS system has already two workflow systems. The OpenACS 4.5 acs-workflow package is a Petri net based Workflow implementation, which is deprecated in OpenACS but used some OpenACS application such as Project Open [16]. Lars Pind, the author of the original workflow package rewrote the original workflow package to reduce its complexity and developed the OpenACS workflow package [17], which is strictly based on finite state machines. While the basic execution mechanism between OpenACS workflow package and XoWiki Content Flow is quite similar, the used framework is very different. XoWiki Content Flow is object oriented (states, actions and conditions can be subclassed) and fully integrated with the content repository (workflow definitions, forms, instances are stored in the content repository with revisions). This allows for redoing certain actions by making an earlier revision of the workflow instance the current revision. Furthermore XoWiki Content Flow is tightly integrated with XoWiki (in particular with XoWiki forms), such that (a) it inherits all properties of a wiki and that (b) it is possible to develop applications with application specific attribute sets just via the web interface. The XoWiki Content Flow package is available via `git://alice.wu-wien.ac.at/xowf`.

References

1. Neumann, G.: XoWiki – towards a generic tool for web 2.0 applications and social software. In: OpenACS and .LRN Spring Conference, International Conference and Workshops on Community Based Environments, Vienna (April 2007)
2. Neumann, G.: Xowiki documentation. `http://media.wu-wien.ac.at/download/xowiki-doc/`
3. Neumann, G., Zdun, U.: XOTcl, an object-oriented scripting language. In: Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference, Austin, Texas (February 2000)
4. Neumann, G., Zdun, U.: XOTcl home page. `http://www.xotcl.org`
5. Neumann, G.: Development the oo-framework for openacs: Improving scalability and applicability. In: International OpenACS and DotLRN Conference: International Conference and Workshops on Community Based Environments, Guatemala (February 2008)
6. Neumann, G., Sobernig, S.: Learning XoWiki: A tutorial to the xowiki toolkit. In: Tutorial at the International OpenACS and DotLRN Conference: International Conference and Workshops on Community Based Environments, Guatemala (February 2008)
7. Bracha, G., Cook, W.: Mixin-based inheritance. In: Proc. of OOPSLA/ECOOP'90. Volume 25 of SIGPLAN Notices. (October 1990) 303–311
8. Neumann, G., Zdun, U.: Enhancing object-based system composition through per-object mixins. In: Proceedings of Asia-Pacific Software Engineering Conference (APSEC), Takamatsu, Japan (December 1999)
9. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases **14**(3) (July 2003)
10. Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, Cambridge, UK (1999)
11. Milner, R.: A Calculus of Communicating Systems. volume 92 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1980)
12. van der Aalst, W.: Pi calculus versus petri nets: Let us eat humble pie rather than further inflate the pi hype. BPTrends **3**(5) (May 2005)
13. Manna, Z., Pnueli, A.: The temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag, Berlin (1992)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
15. Neumann, G., Erol, S.: From a social wiki to a social workflow system. In: post-proceedings of BPM 2008 - 1st Workshop on BPM and Social Soaftware, Milan, Italy (September 2008)
16. Bergmann, F.: Project Open home page. `http://www.project-open.org/`
17. Pind, L.: Package developer's guide to workflow. `http://cvs.openacs.org/cvs/openacs-4/packages/workflow/www/doc/develope%r-guide.html?revision=1.3`