Language-Driven Development of Web-Based Learning Applications

José-Luis Sierra, Baltasar Fernández-Manjón, and Alfredo Fernández-Valmayor

Dpto. Ingeniería del Software e Inteligencia Artificial. Fac. Informática. Universidad Complutense de Madrid
C/ Profesor José García Santesmases s/n. 28040 Madrid. Spain
{jlsierra,balta,valmayor}@fdi.ucm.es

Abstract. In this paper we propose a language-driven approach for the high-level design of web-based learning applications. In our approach we define a domain-specific language that characterizes the key application aspects. Then we assign a suitable operational semantics to this language, and we keep it independent of low-level implementation details such as interaction / presentation or database updating. The resulting design can be easily implemented using the model-view-controller pattern that is very well supported by standard implementation technologies. In addition, these language-driven designs also allow for rapid prototyping, exploration and early discovery of application features, as well as for rational collaboration processes between instructors and developers. We exemplify our approach with a Socratic Tutoring System.

Keywords: Development of web-based learning applications, language-driven development, domain-specific languages document-oriented approach, Socratic tutors.

1 Introduction

As a result of our previous experiences in the development of several web-based E-Learning applications for various purposes [11,18,19,21] we have realized the importance of adopting a well-principled and rigorous approach for modeling the key application aspects (e.g. structure, behavior, interaction) in the very first stages of development. For this purpose, we have adopted a *language-driven approach* [12]. In this approach, we start by defining the *domain specific language* (DSL) that characterizes the key application aspects. Then we assign operational semantics suitable to such a DSL in order to achieve a high-level behavioral characterization of the application. Finally we isolate these semantics of the low-level implementation details regarding the basic interaction, presentation and updating operations. The resulting high-level designs can be easily implemented using the well-known model-view-controller (MVC) pattern [10], which is typically used for organizing almost all modern web-based applications. In addition, this linguistic approach also facilitates rational collaboration processes between instructors and developers. The DSL is near the knowledge and the expertise of the instructors. Therefore they can understand and use the language (provided that a user-friendly notation be available). In this paper we present this language-driven approach.

H. Leung et al. (Eds.): ICWL 2007, LNCS 4823, pp. 520 – 531, 2008.
© Springer-Verlag Berlin Heidelberg 2008
Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

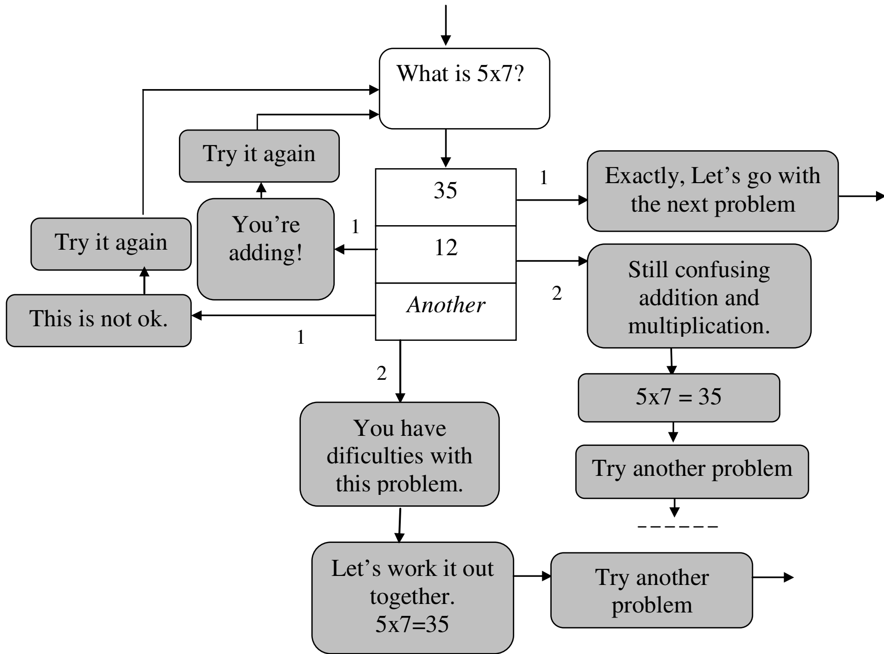


Fig. 1. Graphic representation of a tutorial fragment for a Socratic tutoring system (example adapted from [8])

The structure of the paper is as follows. In section 2 we present a simplified example of the E-Learning application that we will use to illustrate the different aspects of our approach. Section 3 deals with the structural aspects of the design. Section 4 addresses the specification of behavior. Section 5 provides some implementation guidelines. Finally, section 6 presents the conclusions and some lines of future work.

2 An Example of Application

In this paper we will use a simple tutoring system to illustrate the main aspects of our approach. The system is called <e-Tutor>, and it is the web-based version of a previous desktop system developed to explore some concerns in our *document-oriented approach* to the production and maintenance of content-intensive applications [17].

Tutoring systems were popularized during the eighties and nineties of the past century [22]. Although their pedagogical adequacy as mechanisms to support sophisticated learning processes has been heavily questioned, today there is a very active community working in this field, as well as relevant initiatives (see, for instance, [23]). However, our reason for choosing this kind of systems for exemplifying our approach is not so much pedagogical as technological, since the goal of our work is not to criticize or to defend a particular learning approach, but to provide guidelines that can be effectively used to produce and maintain E-Learning applications. For this purpose, we need a language simple enough to be fully addressed in this paper, and

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

this type of tutoring system will let us do so. Indeed, our <e-Tutor> system is a very simplified version of a Socratic Tutoring System, which is based on the seminal work of Prof. Bork's team during the eighties [2,8].

In our <e-Tutor> system the learner will follow tutorials, facing problems whose solution will be constructed by means of a master-disciple dialog. The system will ask questions to the learner and, depending of his/her responses, it will provide him/her with some feedback and it will determine the next step in the learning process. Although in more realistic systems the feedback could depend on the whole history of the previous answers, in our simplified version it will depend exclusively on the number of times that the learner gives a particular answer to a question. In Fig. 1 we show a simple example of this kind of organization for the tutorials.

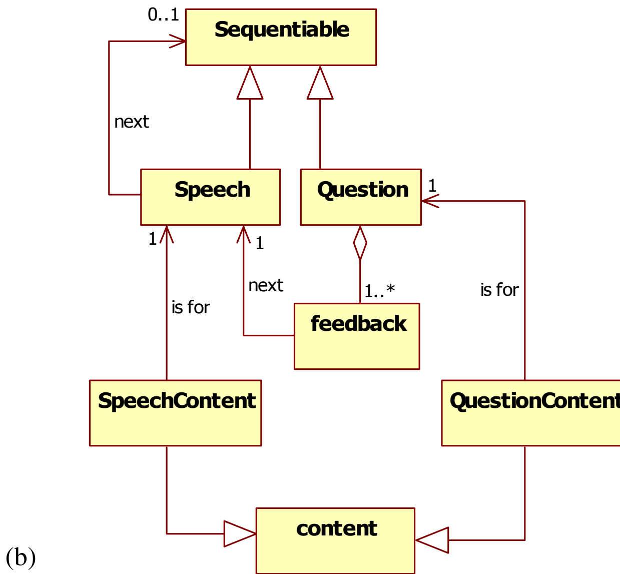
3 Designing the Language

The first step in our approach is to design a language for describing the key application aspects. It is important to point out that this language does not try to represent the *actual* application's features (e.g. the actual messages and hints provided to the learner), but only to represent their structure. Therefore, this language will include mechanisms to refer to the actual information. In this sense, the language is in some way similar to the languages used for marking documents up [5]. Furthermore, the design of this language will be addressed from an abstract point of view. Thus, the language itself will be characterized as an *information model*, instead of as a grammar for a concrete textual or visual language. This information model can be provided using standard data modeling techniques (e.g. entity-relationship diagrams, or UML class diagrams), and it can be further formalized in terms of a first-order signature (i.e. a set of function and predicate symbols) with the aim of subsequently enabling the formal definition of the operational semantics.

In the case of <e-Tutor>, the abstract structure of the Socratic dialog in the previous section is a weighted directed graph whose nodes contain different types of information, and which is oriented to represent the type of dialog between master and disciple on which this kind of tutorials is based. In Fig. 2a we sketch the information model for the <e-Tutor> language. In Fig. 2b we further formalize this model in terms of predicate symbols. We also document them with their intended meanings.

Notice that our approach follows the current practice in E-Learning specifications, where each specification is usually accompanied by its corresponding information model. Concrete notations can then be subsequently provided as appropriate *bindings* (e.g. as XML-based markup languages). Notice that it is also a common practice in the design of conventional computer (usually programming) languages, where abstract and concrete syntaxes are clearly identified. Abstract syntaxes let language designers capture the features of the language that are essential for further specifying the meaning of their constructs, while concrete syntaxes provide notations for the final language's users [7]. In our approach concrete syntaxes (*bindings*, following the more widely accepted terminology in E-Learning) will be very important in order to involve instructors in the production and maintenance of the learning contents and other pedagogical aspects. Indeed, we will promote the provision of bindings as

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information



(b)

Predicate	Intended meaning
speech(s,n)	The speech s is followed by n. In this item, n can be another speech, a question point, or the end of the learning process. Besides, s is a unique identifier.
question(q)	A question point identified with the unique identifier q.
feedback(q,a,n,s)	A feedback for the answer a (a unique identifier) collected in a question point q. Besides, n is the number of times that the answer has been collected. The feedback itself will start with speech s.
content(i,c)	The content of the question / speech i is c.

Fig. 2. (a) Information model for the <e-Tutor> language; (b) predicates used to formalize the model in (a)

domain-specific descriptive markup languages (i.e. we promote DSLs with XML bindings) for the application's contents and complementary aspects, as proposed in our work on the aforementioned document-oriented approach.

4 Specifying the Operational Semantics

Once the language for structuring the application is available, developers must model the runtime behavior of such a language by assigning to it suitable *operational semantics*, which model such a behavior as transitions between computation states [14]. Therefore, we firstly need to decide on a suitable representation for the computation

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

states. Next we must characterize the transition relation that governs how to go from one state to another by using semantics rules. These aspects are detailed in the next sections.

4.1 Representing the Computation States

Computation states will characterize the internal state of an abstract machine that executes the devised language. Therefore, typically such states will include:

(a)

Term	Intended meaning
answered (*a*)	The learner has given the answer *a* to the last question.

(b)

Term	Intended meaning
do-start	The system execution is starting.
do-speech(*s*)	The system is proffering the speech *s*
do-ask(*s*,q)	The system, by proffering the speech *s*, is asking the question *q*.
do-end	The system execution is ending.

Fig. 3. (a) Terms in the *view2controller* stream; (b) Terms in the *controller2view* stream

- The application description, in terms of the language provided. This description will play the role of the *program* stored in the machine's memory. Alternatively, sometimes it will be possible to drop this description from the computation states themselves, and to assume that it is globally available.
- A *control term*, which will usually coincide with the predicate definition in the application description currently under consideration. This term will be used to decide the next step to take in the execution.
- A *context*, which contains additional information used to perform the transitions and which expresses the dependence of the actions on previous responses or states.
- Three different streams. The *view2controller* stream will contain suitable representations of the user's interactions. In the *controller2view* stream the controller will write appropriate commands to govern the view's update. Finally, in the *controller2model* stream the controller will write appropriate update commands for the information model (for those applications where the model is not updated, but only queried, this last stream can be safely omitted). These streams are very useful for isolating the behavioral details from the presentation / interaction / updating aspects. Also this organization naturally leads to an MVC architecture for the final implementation, as previously mentioned.

In <e-Tutor>, computation states will be associated with the states of the master-disciple dialog and the evolution of this dialog. Therefore we will represent these computation states as 5-tuples of the form $<\theta,T,\rho,in,out>$, where:

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

- θ is the control term and T is the structure of the tutorial represented as a set of ground (i.e. variable-free) facts using the signature described in the previous section.
- ρ is the context, which in this example will be constituted by a set of counters associated with the answers. With ρ_C we will denote the value of the *c* counter in ρ, while with $\rho_C := v$ we will denote the new set of counters resulting from updating the value of *c* in ρ with *v*. Also, if a counter is not set, its value will be considered 1, since it will be the first time that the learner proffers the associated answer.
- *in* and *out* are the *view2controller* and the *controller2view* streams respectively. Since in this example the model is not updated, the *controller2model* stream is not required, and it can be dropped from the computation state. In Fig. 3a we characterize the possible terms in the *view2controller* stream, which will simply contain the learner's answers. In Fig. 3b we show the possible terms in the *controller2view* one, which will contain the commands for presenting the system's speeches and questions, as well as for announcing the init and the end of the system's execution.

Besides this kind of states, we will also introduce a special format for initial and final states. Initial states will be represented as $<s,T,in,out>$, with *s* the starting speech. Final states will be represented simply as $<\rho,out>$.

4.2 Describing the Semantic Rules

Once we agree on the structure of the computation states, we can formalize the language's runtime behavior. For this purpose we propose using the *structural style* of operational semantics [14,16]. In this style the semantics will be described by a set of *semantic rules* that will resemble the rules of formal *calculi* in logic. These rules characterize the transitions between computation states. With $\sigma \rightarrow \sigma'$ we will denote the transition from the state σ to the state σ'. In these rules we also can use additional constraints expressed with logic and set-theoretic notations. For this purpose, we will use $\vdash\Phi$ to denote a logical / set-theoretical formula Φ that must be true. At the top of the rules we put the applicability conditions, using such logical notations, while at the bottom we describe the resulting transitions. In addition, we encourage the use of a *small-step* style of specifying the semantics [14]. This style concentrates on characterizing transitions between consecutive states, and will ease the move to a subsequent implementation.

In Fig. 4 we show the semantic rules for our example. Thus, these rules formally state the informal behavior outlined in section 2. In these rules stream manipulation is abstracted using the *in* and the *out* operations, whose behavior is left unspecified. The rules themselves read as follows:

- The *starting* rule models the execution's init. For this purpose, the starting speech is queried in the tutorial. Besides, the first speech is picked from such a tutorial and it is set as the control term. Also notice that the context with the answers' counters is initialized to the empty set (i.e. as indicated above, the counter for every answer will be 1). Finally, a suitable command announcing the beginning of the execution is written in the *controller2view* stream.

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

- The *speaking* rule models how the system proffers a speech which is followed by another speech: a command to proffer the speech is written in the *controller2view* stream, and the next speech is set as the control term.
- The *asking* rule models the proffering of a speech when it is followed by a question point. This time the system announces the question asked with the speech to be proffered.
- The *evaluating* rule models what happens at a question point. The learner's answer is read from the *view2controller* stream, a suitable feedback is picked from the tutorial, and its associated speech is retrieved and set as the control term. Besides, the counter associated with the answer is incremented.
- Finally, the *ending* rule models how the system finishes the execution. It holds when there is not such a thing as a following question or speech in the learning flow. Hence the final speech is proffered, and a command announcing the end is written in the resulting output stream.

$$\frac{\vdash \mathrm{speech}(s,ns) \in T}{\langle s,T,In,Out\rangle \rightarrow \langle \mathrm{speech}(s,ns),T,\varnothing,In,\mathrm{out}(Out,\text{do-start})\rangle} \quad \textbf{starting}$$

$$\frac{\vdash \mathrm{speech}(ns,nns) \in T}{\langle \mathrm{speech}(s,ns),T,\rho,In,Out\rangle \rightarrow \langle \mathrm{speech}(ns,nns),T,\rho,In,\mathrm{out}(Out,\text{do-speech}(s))\rangle} \quad \textbf{speaking}$$

$$\frac{\vdash \mathrm{question}(q) \in T}{\langle \mathrm{speech}(s,q),T,\rho,In,Out\rangle \rightarrow \langle \mathrm{question}(q),T,\rho,In,\mathrm{out}(Out,\text{do-ask}(s,q))\rangle} \quad \textbf{asking}$$

$$\frac{\vdash \mathrm{feedback}(q,a,\rho_a,s) \in T \;;;\; \vdash \mathrm{speech}(s,n) \in T \;;; \vdash \mathrm{in}(In) = \langle \mathrm{answered}(a), In'\rangle}{\langle \mathrm{question}(q),T,\rho,In,Out\rangle \rightarrow \langle \mathrm{speech}(s,n),T,\rho_a := \rho_a + 1, In', Out\rangle} \quad \textbf{evaluating}$$

$$\frac{\vdash \mathrm{speech}(n,_) \notin T \;;;\; \vdash \mathrm{question}(n) \notin T}{\langle \mathrm{speech}(s,n),T,\rho,_,Out\rangle \rightarrow \langle \rho,\mathrm{out}(\mathrm{out}(Out,\text{do-speech}(s)),\text{do-end})\rangle} \quad \textbf{ending}$$

Fig. 4. Semantic rules for the <e-Tutor> language

```
transition([S,In,Out]->[speech(S,Ns),[],In,NOut]) :-
    out(Out,do_start,NOut),
    speech(S,Ns).

transition([speech(S,Ns),Cs,In,Out]->
               [speech(Ns,NNs),Cs,In,NOut]) :-
    speech(Ns,NNs),
    out(Out,do_speech(S),NOut).

transition([speech(S,Q),Cs,In,Out]->
               [question(Q),Cs,In,NOut]) :-
    question(Q),
    out(Out,do_ask(S,Q),NOut).
...
```

Fig. 5. Encoding of the *starting*, *speaking* and *asking* rules in a Prolog prototype for the <e-Tutor> language

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

We have realized how the effort employed in this kind of specifications pays out, since these specifications are very valuable in anticipating the more obscure aspects of the application's dynamic behavior without being obfuscated by technological and/or implementation details. Indeed, based on these specifications it is possible to perform rapid prototyping of the applications using, for instance, approaches similar to those described in [6]. As an example, in Fig. 5 we show a fragment of a Prolog prototype for the semantics of Fig. 4 (see [20] for more details). Finally, the formal flavor of the specifications provides the opportunity to apply optimizations and refinements in very early stages of the development process.

5 Implementation

In this section we examine how to organize the final applications from their designs in the terms stated in this paper. While the particular implementation strategies depend on the particular technologies and platforms chosen, we can still abstract some general implementation guidelines that can be useful in most of the cases. As indicated above, the resulting applications are amenable to being architected following the MVC pattern. Semantic rules in the language's operational semantics are useful in order to structure the application's controller. The language's structural characterization is in turn useful to structure the application's model. Finally, the structure of the different streams involved in the semantics is useful to identify the basic presentation commands, user actions and updating operations.

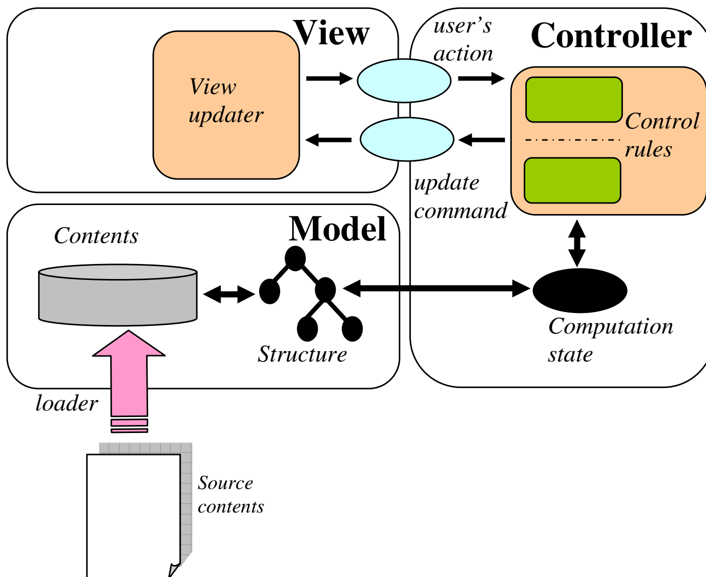


Fig. 6. A possible organization for the final implementations

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

In Fig. 6 we sketch the resulting organization. Notice how the computation state is configured as a global structure that in turn makes reference to information elements in the model (the implementation counterparts to the ground facts used to control the language's runtime behavior in the operational semantics). Also notice how the stream-based communication between the controller and the view is refined using call-back mechanisms. For this purpose, the controller is activated as response to the user's events. As a consequence, the view is updated. This updating is centralized using a *view updater* component. Communication itself is carried out using appropriate *user's actions* and *update commands*, which are transferred between the controller and the view. Finally, also notice how contents are encapsulated in the model. The updating of these contents is performed by invoking the appropriate operations on the model's structure, which in turn will act on the actual contents. Also notice that this structure can be referenced in the user's actions and in the update commands, therefore making the relevant parts of the model accessible for the view. Finally, notice how a *loader* is explicitly identified. It lets us load bodies of author-oriented contents in the application's model.

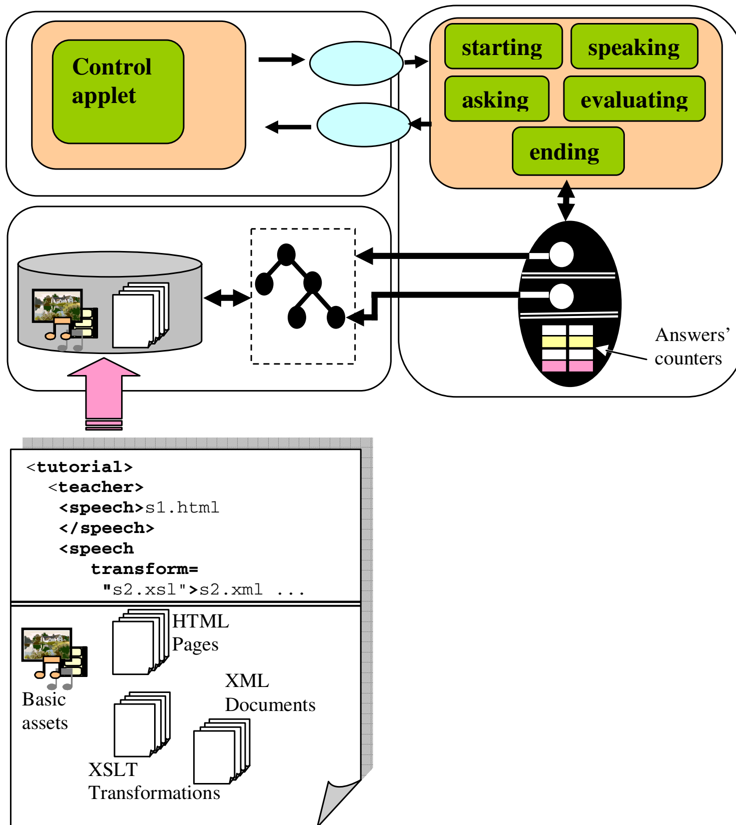


Fig. 7. High-level organization of the web-based version of <e-Tutor>

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

These contents will be structured according to a suitable binding of the application DSL. In our experiences we have performed such bindings in terms of appropriate descriptive markup DSLs, letting authors provide the contents as documents structured according to such languages [17].

The structure of the web-based version of <e-Tutor> is sketched in Fig. 7. The controller is organized in terms of the semantic rules sketched in the previous section. Since communication with the view is managed in terms of an observer-observable base, the streams are dropped from the computation state. This state maintains a reference to the overall model's structure, another reference to the current control term, and a table with the answers' counters.

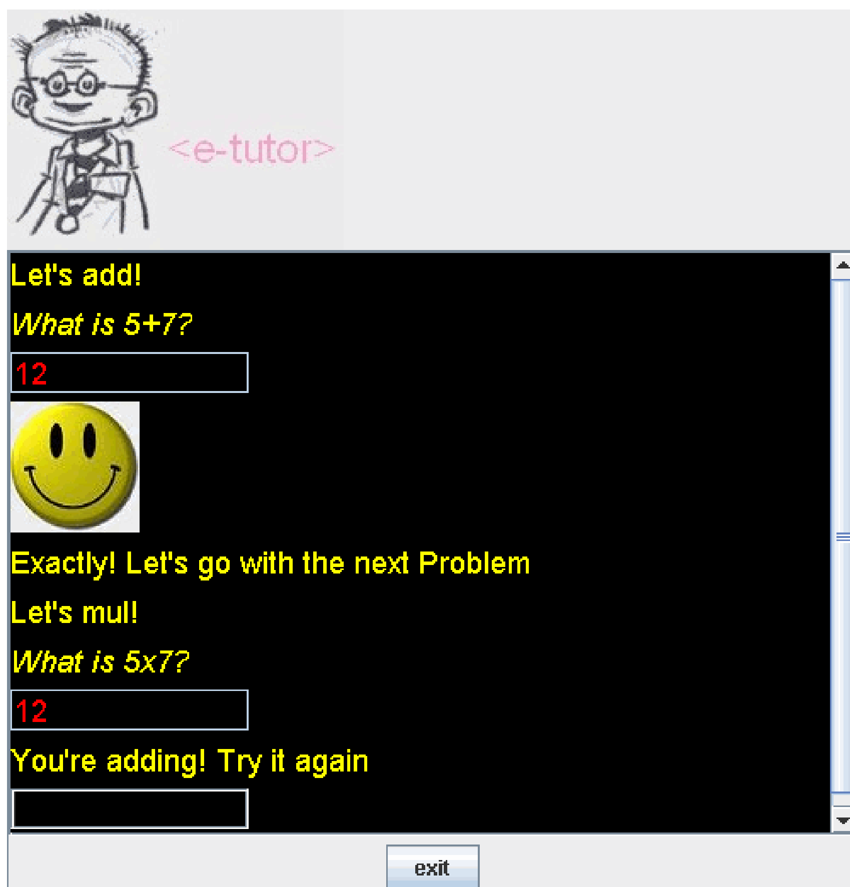


Fig. 8. Snapshot for a run of the simple tutorial of Figure 1 in <e-Tutor>

We use a java applet to implement the *view updater*. Indeed, by inspecting the operational semantics we can discover the proactive nature of the controller in a very early stage of the development process. It means that once the controller has read a user's action, it can communicate to the view several update commands, since the

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

tutoring system can proffer several speeches before asking a new question. Since this behavior is not directly supported in HTTP, we need an active component on the client side (in this case, the control applet) taking care of it. Indeed, this type of internet-rich applications is very common in web-based learning, where similar solutions have also been adopted (e.g. the runtime in the SCORM – Shareable Content Object Reference Model – specification [1]) and the rigorous design of the application can help to anticipate it. Currently we are re-factoring this implementation in terms of AJAX technology [15]. Regardless of the migration of the controller aspect to the client side, the implementation is being substantially facilitated by the earlier effort made in the formal design, which has been shown to be pretty independent of the subsequent implementation platform or technology.

Finally, the structure of the model is a direct implementation of the language's relational structure. Contents themselves are organized as a set of basic multimedia assets (e.g. images, videos, sounds, etc.) and HTML pages. In order to produce all these components, we introduce a user-friendly XML-based markup language [3], which can be used by instructors to structure the tutorials. The resulting XML documents are then processed to produce the abstract representations of the structure of the tutorials. Instructors can prepare the actual contents directly as HTML pages, but they can also use other domain-specific XML-based markup languages. The resulting XML documents can be transformed into final HTML presentations using suitable XSLT transformations [4].

In Fig. 8 we show a snapshot of the tutorial corresponding to the example in Fig. 1.

6 Conclusions and Future Work

In this paper we have presented a language-driven approach to the high-level design of web-based learning applications. The resulting applications are easily architected according to the MVC pattern. The approach is focused on characterizing the key application aspects with a language specifically designed for such a kind of application (i.e. a DSL). Then the interactive behavior is described with appropriate operational semantics for this language. As it has been tested in several developments, this approach promotes an innovative way of collaboration between instructors and developers during the design and development of E-Learning applications. It also facilitates rapid prototyping, as well as the discovering of relevant features of the interactive behavior of the final applications in very early stages of the development.

Currently we are systematizing the approach and further testing it in many other scenarios [13]. In particular, we hope to apply the approach to systems driven by educational modeling languages such as those described in [9].

Acknowledgements

The Spanish Committee of Education and Science (Projects TIN2004-08367-C02-02 and TIN2005-08788-C04-01) and the Regional Government / Complutense University of Madrid (research group 910494) have partially supported this work.

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

References

1. Advanced Distributed Learning - Shareable Content Object Reference Model (ADL-SCORM), Faulkner Information Services (2003)
2. Bork, A.: Personal Computers for Education. Harper & Rows, New York (1985)
3. Bray, T., et al. (eds.): Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation (2000)
4. Clark, J. (ed.): XSL Transformations (XSLT) Version 1.0. W3C Recommendation (1999)
5. Coombs, J.H., Renear, A.H., DeRose, S.J.: Markup Systems and the Future of Scholarly Text Processing. Communications of the ACM 30(11), 933–947 (1987)
6. Clément, D., et al.: Natural Semantics on the Computer. Tech. Rep. 416. INRIA (1985)
7. Friedman, D., Wand, M., Hayes, C.T.: Essentials of Programming Languages, 2nd edn. MIT Press, Cambridge (2001)
8. Ibrahim, B.: Software Engineering Techniques for CAL. Education & Computers 5, 215–222 (1989)
9. Koper, R., Tatersall, C. (eds.): Learning Design: A Handbook on Modeling and Delivering Networked Education and Training. Springer, Heidelberg (2005)
10. Krasner, G.E., Pope, T.S.: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk 80 System. Journal of Object Oriented Programming 1(3), 26–49 (1988)
11. Martínez-Ortíz, I., Moreno-Ger, P., Sierra, J.L., Fernández-Manjón, B.: <e-QTI>: A Reusable Assessment Engine. In: Liu, W., Li, Q., Lau, R.W.H. (eds.) ICWL 2006. LNCS, vol. 4181, pp. 134–145. Springer, Heidelberg (2006)
12. Mauw, S., Wiersma, W.T., Willemse, T.A.C.: Language-driven System Design. International Journal of Software Engineering and Knowledge Engineering 14(6), 625–664 (2004)
13. Moreno-Ger, P., Sierra, J.L., Martínez-Ortiz, I., Fernández-Manjón, B.: A Documental Approach to Adventure Game Development. Science of Computer Programming 67(1), 3–31 (2007)
14. Mosses, P.D.: Formal Semantics of Programming Languages: An Overview. Electronic Notes in Theoretical Computer Science 148(1), 41–73 (2006)
15. Paulson, L.D.: Building Rich Web Applications with AJAX. IEEE Computer 38(10), 14–17 (2005)
16. Plotkin, G.D.: An Structural Approach to Operational Semantics. Technical Report DAIMI FN-19. Computer Science Dept. Aarhus University (1981)
17. Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B.: A Document-Oriented Paradigm for the Construction of Content-Intensive Applications. Computer Journal 49(5), 562–584 (2006)
18. Sierra, J.L., et al.: From Research Resources to Virtual Objects: Process model and Virtualization Experiences. Journal of Educational Technology & Society 9(3), 56–68 (2006)
19. Sierra, J.L., et al.: A Highly Modular and Extensible Architecture for an Integrated IMS based Authoring System: The <e Aula> Experience. Software-Practice & Experience 37(4), 441–461 (2007)
20. Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B.: How to Prototype an Educational Modeling Language. In: Proc. of the IX International Simposium on Computers in Education SIIE 2007, November 14–16, 2007, Porto, Portugal (2007)
21. Sierra, J.L., Moreno Ger, P., Martínez Ortiz, I., López Moratalla, J., Fernández-Manjón, B.: Building Learning Management Systems Using IMS Standards: Architecture of a Manifest Driven Approach. In: Lau, R.W.H., Li, Q., Cheung, R., Liu, W. (eds.) ICWL 2005. LNCS, vol. 3583, pp. 144–156. Springer, Heidelberg (2005)
22. Sleeman, D., Brown, J.S. (eds.): Intelligent Tutoring Systems. Academic Press, London (1982)
23. XTutor web site. `http://icampus.mit.edu/xtutor` (last visited June 8, 2007)

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information