# From Documents to Applications Using Markup Languages

**José Luis Sierra, Alfredo Fernández-Valmayor, and Baltasar Fernández-Manjón,**
*Complutense University of Madrid*

This document-oriented approach to developing content-intensive applications uses markup languages to involve domain experts in development and to simplify application production and maintenance.

In typical publishing scenarios, descriptive markup languages let authors describe a document's logical structure without compromising its processing. During the last 10 years, we've realized how these languages can also play a critical role in developing *content-intensive* applications such as hypermedia and educational applications and knowledge-based systems. These kinds of applications typically require large amounts of highly structured information and development processes that involve application domain experts at almost every stage.

Our document-oriented approach for developing content-intensive applications[1] combines the main ideas of software development based on domain-specific languages[2] with the separation of concerns in descriptive markup languages (see the "Descriptive Markup Languages" sidebar). We illustrate this approach with <e-tutor>, an application we built in the educational domain.

## The document-oriented approach

This approach has three main steps:

- Developers collaborate with domain experts to formalize a suitable domain-specific descriptive markup language using a document grammar and to decide on the language's features.
- Domain experts describe the desired application as a marked-up application document comprising the application's contents and other features. As domain experts edit the application documents, they use the document grammar to validate its structure.
- Developers produce the application kernel—a software artifact that, when fed with the marked-up document, yields the final application.

So, by creating and manipulating documents, the experts are actually in charge of designing and maintaining the applications. The developers in turn take care of the corresponding linguistic and operational support: formalizing the markup languages and constructing the application kernels. This approach promotes the involvement of domain experts in producing content-intensive applications. Indeed, it promotes the design of markup languages that are tailored to the domain experts' particular expertise and skills and that closely mirror the structure of the documents that the experts manage during their daily work. You can find other related approaches in the "Related Work in Document-Oriented Approaches" sidebar.

### The <e-tutor> environment

Our <e-tutor> environment is an environment for developing Socratic tutoring systems as desktop applications.[3] In a Socratic tutoring system, learners


0740-7459/08/$25.00 © 2008 IEEE

## Descriptive Markup Languages

Markup is the text that we add to a document to convey information about it. When the markup identifies a document's logical structure instead of representing a processing instruction or formatting command, we call it *descriptive*. We can represent this logical structure using descriptive *tags* that delimit the document's *logical elements*. These elements can also have *lexical attributes* attached that convey additional information. Because elements can in turn contain other elements, the structure of a document that we represent with descriptive markup is usually tree-like.

A *descriptive markup language* is a set of rules that descriptively govern the markup of documents of a particular type. We can use metalanguages such as SGML or XML when we define a descriptive markup language. These metalanguages let us formally define the syntax of a particular descriptive markup language using a document grammar. They also govern how to apply the markup to the document data. SGML and XML incorporate a built-in grammatical formalism called a DTD (document type definition).

XML practitioners have defined many other alternative, more powerful formalisms for document grammars. One example is XML schema.

Further readings on this topic are available elsewhere.[1–3]

### References


1. J.H. Coombs, A.H. Renear, and S.J. DeRose, "Markup Systems and the Future of Scholarly Text Processing," *Comm. ACM*, vol. 30, no. 11, 1987, pp. 933–947.
2. C.F. Goldfarb, "A Generalized Approach to Document Markup," *ACM SIGPLAN Notices*, vol. 16, no. 6, 1981, pp. 68–73.
3. A. Renear, E. Mylonas, and D. Durand, "Refining Our Notion of What Text Really Is: The Problem of Overlapping Hierarchies," *Research in Humanities Computing*, S. Hocky and N. Ide, eds., Oxford Univ. Press, 1986.


## Related Work in Document-Oriented Approaches

These initiatives have influenced our document-oriented approach:

- HyTime, a sophisticated SGML application for producing hypermedia: ISO/IEC Standard 10744, *Hypermedia/Time-based Structuring Language* (*HyTime*), 2nd ed., ISO, 1997.
- The literate programming paradigm, initially proposed by Donald E. Knuth, which combines human-readable documentation with machine-readable source code in a single document: D.E. Knuth, "Literate Programming," *Computer J.*, vol. 27, no. 2, 1984, pp. 97–111.
- Jargons, an approach to software development that conceives of domain-specific languages as markup languages: L.H. Nakatani and M. Jones, "Jargons and Infocentrism," *Proc. 1st ACM SIGPLAN Workshop Domain-Specific Languages* (DSL 97), ACM Press, 1997, pp. 15–24.
- Work in program generation—in particular, J. Craig Cleaveland's work on building program generators with XML and Java technologies: J.C. Cleaveland, *Program Generators with XML and Java*, Prentice Hall, 2001.
- IMS e-learning specifications, such as *Question and Test Interoperability* and *Simple Sequencing*, that provide domain-specific markup languages describing different aspects of an e-learning system: www.imsglobal.org.

build solutions to problems through master-disciple (teacher/learner) dialogues. The system analyzes the learner's responses, then provides feedback. Then, it determines the next step to undertake in the learning process. To adapt the feedback to the individual's learning path, the system counts the number of times the learner gives a particular answer to a question. In our environment, the feedback depends on these counters' values.

We built <e-tutor> by following the three main steps in document-oriented development outlined earlier (see figure 1).

In the first step, the developers collaborate with the instructors (the domain experts in this example) to formalize the <e-tutor> language. This XML-based language makes it possible to mark up the tutoring system's *speech acts*—that is, textual phrases or images illustrating important concepts. The language also lets instructors mark up the *question points* in the dialogue—those points where the system has asked the learner a question. At these question points, the instructors identify an input method to collect the learner's responses (for example, the learner might need to give a numeric quantity). The instructors also anticipate the learner's possible answers, the corresponding feedback, and the next problems that the learner must address. In addition, the language uses elements'

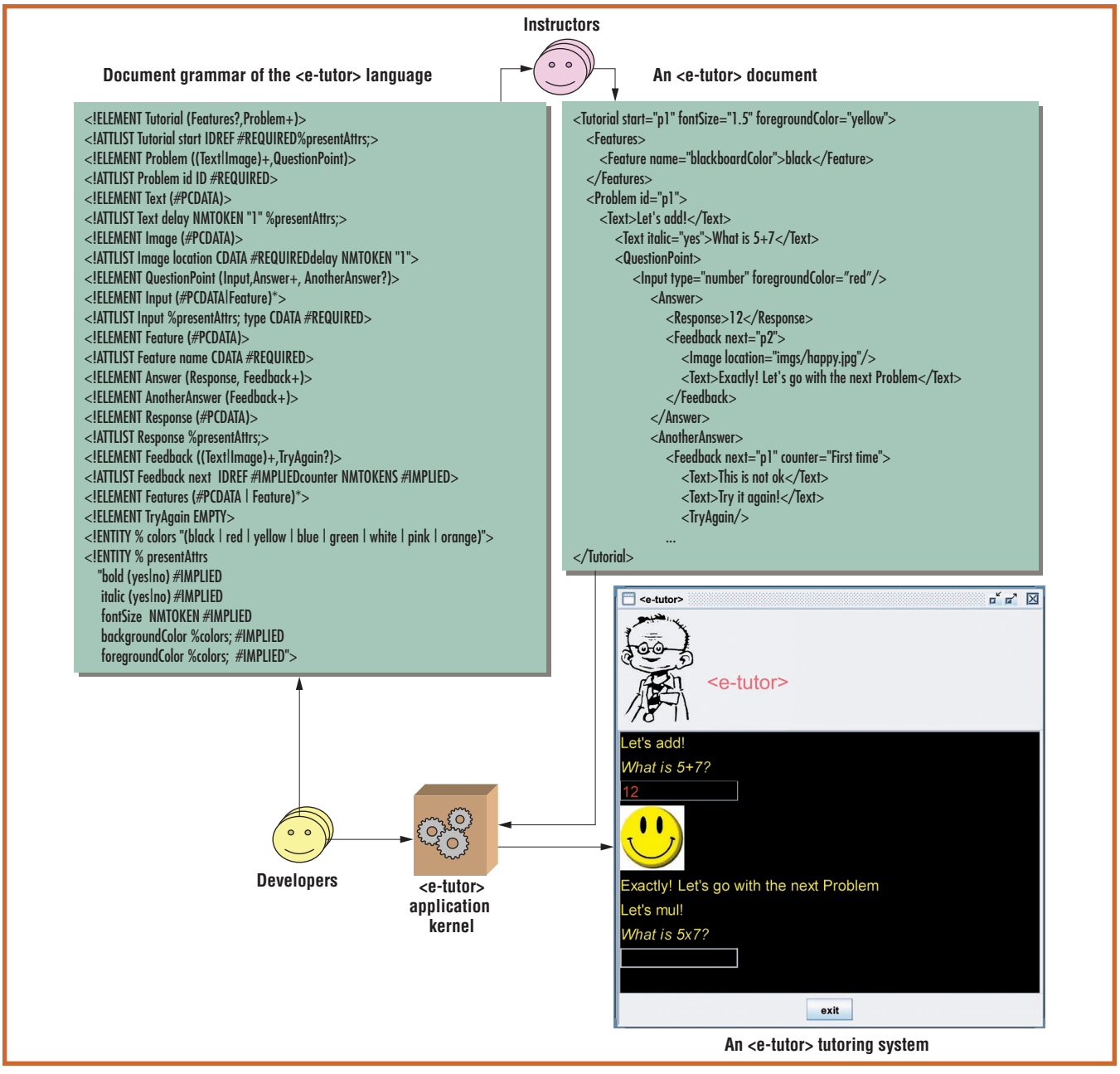


**Figure 1. The document-oriented approach in <e-tutor>.**

lexical attributes to describe other pedagogically relevant application features (for example, the time interval between two speech acts, when an answer is appropriate depending on a counter's value, or presentation data such as the color and size of a speech's font).

In the second step, the instructors use the <e-tutor> language to produce an <e-tutor> document, which describes the tutoring system.

Finally, in the third step, the developers build and maintain an <e-tutor> application kernel. As the kernel acts on an <e-tutor> document, it yields the tutoring system.

Figure 1 summarizes the document-oriented approach in <e-tutor> and outlines the <e-tutor> language's document grammar. For the sake of conciseness, we use an XML DTD (document type definition) instead of an XML schema. This sample also shows a document fragment describing part of a simple tutoring system on elementary arithmetic, which we adapted from Alfred Bork.[3]

## The application kernel

This kernel combines an object-oriented domain-specific application framework with an application generator. The generator processes marked documents and generates the application as an instantiation of the framework. Using a generator is

## Attribute Grammars

Language designers usually use *context-free* grammars to describe the syntax of computer languages. These grammars include a set of syntax rules describing the language's different syntactic constructs. Using a context-free grammar for a language, it's possible to describe the syntactic structure of each sentence in the language with a parse tree for the sentence. The tree's leaves are the primitive syntactic elements in the sentence. Inner nodes correspond to applications of the syntax rules.

*Attribute grammars* extend context-free grammars to allow the description of additional aspects beyond syntax (for example, type constraints or translation into object code). An attribute grammar adds a set of *semantic attributes* to the symbols of the underlying context-free grammar and a set of *semantic equations* to each syntax rule. These equations indicate how to compute some attributes' values (that is, how to evaluate the attributes). For this purpose, equations apply semantic functions to other attributes in the syntax rule. As a consequence, they also introduce some dependencies between attributes: when we use one set of attributes to evaluate another attribute, we say that this attribute depends on that set. All these dependencies let us draw a dependency graph for each parse tree. The attribute evaluation order must be compatible with the dependency graph. Researchers on attribute grammars have developed many techniques to produce evaluators that ensure such compatibility.

For more details, see Donald E. Knuth's original work[1] and Jukka Paakki's survey.[2]

### References


1. D.E. Knuth, "Semantics of Context-free Languages," *Mathematical Systems Theory*, vol. 2, no. 2, 1968, pp. 127–145; correction published in *Mathematical System Theory*, no. 5, no. 1, 1971, pp. 95–96.
2. J. Paakki, "Attribute Grammar Paradigms—A High-Level Methodology in Language Implementation," *ACM Computing Surveys*, vol. 27, no. 2, 1995, pp. 196–255.


similar to using any component driven by XML documents (for example, general-purpose XSLT transformation engines or domain-specific validation engines such as jCAM). We base our generators on the main concepts behind *attribute grammars*, a classical tool in language processing (see the "Attribute Grammars" sidebar). Generators turn application documents into document trees, which explicitly represent the hierarchical structures of these marked-up documents. Then they add semantic attributes to each of the document tree's element nodes (which correspond to the document's logical elements) and provide semantic functions to compute these attributes' values. Finally they generate the applications by evaluating these semantic attributes.

This approach facilitates the incremental construction of the application kernel. Indeed, in any realistic application domain, the application language will evolve to accommodate new features. Therefore, the application kernel must evolve in accordance. We can use good practices in object-oriented software development to manage the application framework's evolution. In turn, the concepts behind the attribute grammar paradigm also facilitate the evolution of the generation part. They let us decompose the generation problem in small, affordable computations on well-defined patterns in the document trees. Besides, because we don't have to make the execution order of these computations explicit, the resulting designs are more declarative and maintainable than a typical ad hoc XML processing program. We can easily extend a design to accommodate new features or new extensions in the application language. We can also add new semantic attributes as well as redefine and extend the computations of the exiting ones.

***The kernel's internals.*** Figure 2 presents an application kernel's internal architecture. The exact nature of the application framework depends on the particular application domain. The generator includes three components:

- The *parser* receives the application language's document grammar and the application document as input and produces the corresponding document tree. Only a valid document with respect to that grammar will describe legitimate applications.
- The *semantic factory* formalizes, in terms of semantic attributes and functions, the meaning of each element that the experts and developers define in the document grammar. Notice the difference between the declarative lexical attributes in the document grammar and the semantic attributes that formalize the application document's operational semantics. Like the application framework, semantic factories are specific to each application domain.
- The *semantic tree builder* receives the document tree and the semantic factory as input and produces a semantic tree made up of semantic nodes, each corresponding to an element node in the document tree. In turn, each semantic node holds a set of semantic attributes. Fi-

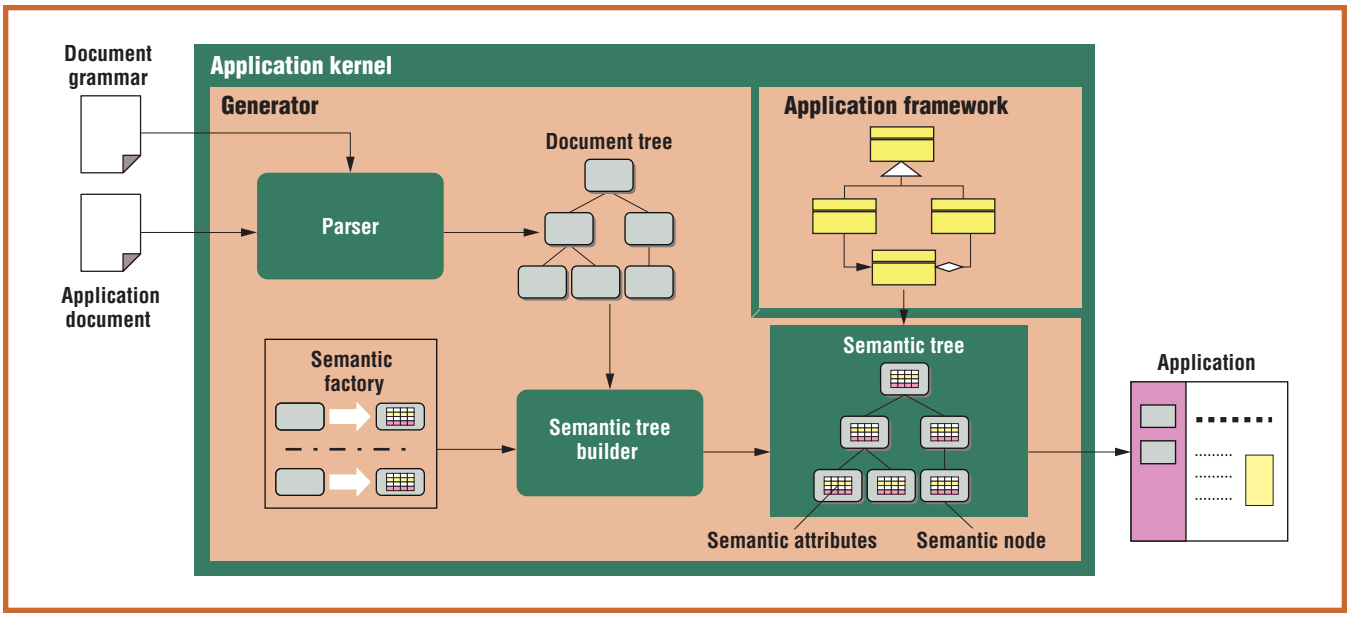


**Figure 2. Architecture of an application kernel.**

nally, each semantic attribute holds a semantic function for computing the attribute's value, a variable for storing its result, and a marker (this marker is "on" when the attribute has a value assigned, and "off" otherwise). Typically, semantic functions perform appropriate instantiation actions on the application framework and return references to the resulting objects. Also, these functions often consult other attributes' values. This supposes an implicit encoding of the dependencies between the semantic tree's semantic attributes.

The goal of the semantic tree is to instantiate the application framework to produce the final application. Indeed, the semantic attributes in such a semantic tree will typically refer to partial fragments of the application during its generation. The application kernel usually requires an attribute value in the tree's root, which will finally refer to an object representing the whole application. The attribute evaluation process will take care of the rest. In this process, when a semantic node receives the request to get a semantic attribute's value, the node does one of two things:

- If the marker is on, it obtains the variable's content.
- If the marker is off, it invokes the semantic function, then stores the resulting value in the variable, and turns the marker on. This prevents the reevaluation of the semantic function if the node receives the request again.

Application kernels reuse the same parser and semantic tree builder. So, to build a particular application kernel, developers must only provide a suitable implementation of the application framework, a suitable semantic factory defining the document grammar's operational semantics, and a main program gluing everything together.

***The generation framework.*** In constructing application kernels, developers can benefit from using a suitable framework for developing generators. Based on our previous experiences with the document-oriented approach, we built one such framework with Java as the implementation language. We constructed it on top of DOM (the W3C Document Object Model specification for the tree-based processing of XML documents) and JAXP (the Java API for XML Processing, which lets us connect with an underlying XML parsing framework in a transparent way). We also used Java's reflection API to facilitate the development of semantic factories. Figure 3 depicts the framework's most relevant components, including external ones (shown in orange). For the sake of conciseness, we omit details concerning package organization and exception handling as well as other minor accessory classes.

The framework lets developers implement semantic functions as Java methods and package them in a Java class that we'll call a *semantic module*. They must annotate these methods with `ForAttribute` annotations (the `elements` property identifies the set of elements to which the method applies, and the `attribute` property the semantic attribute). They also can derive these modules from the `BaseSM` class to make all the contextual information of the corresponding semantic node available to those meth-

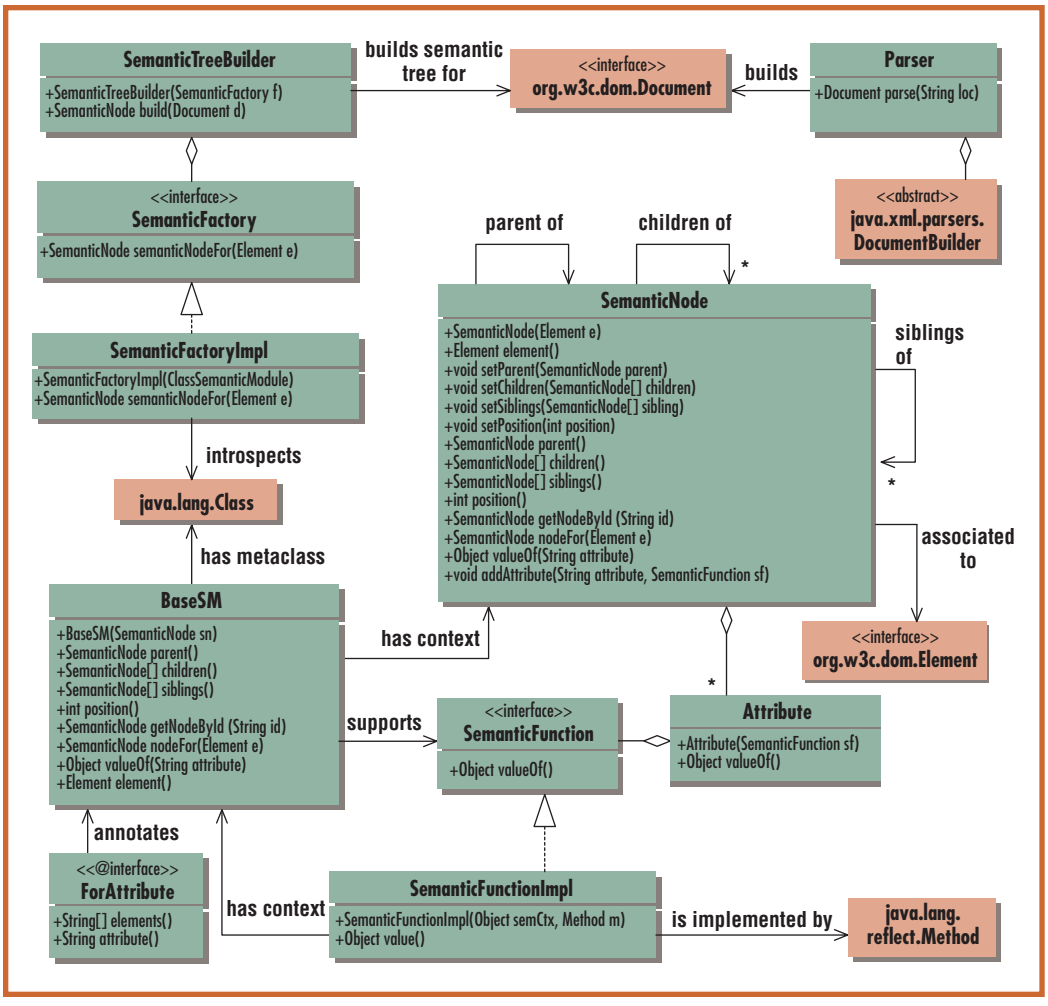


**Figure 3. The generation framework's main components.**

ods. Then they must turn this semantic module into a semantic factory, which must follow the SemanticFactory interface. For this purpose, they can use the SemanticFactoryImpl class, which provides the appropriate reflective support. In providing the rest of the generator, they can use the Parser and the SemanticTreeBuilder classes. The Parser class implements the parser component in terms of a JAXP's DocumentBuilder, the basic JAXP component for building DOM trees. The SemanticTreeBuilder class implements the semantic tree builder component, and, besides building the semantic tree, it registers the corresponding semantic node as a user property of each element node. This makes the semantic node accessible to any other component.

The rest of the components represent semantic trees. We represent the nodes in these trees with the SemanticNode class. This class includes an operation for consulting the associated element (element method). This also includes operations to get and set the semantic nodes associated with the parent, the child and sibling elements, as well as the node's position in the array of siblings. It lets semantic functions access the semantic information of the parent, child, and sibling nodes in a straightforward way. As in many implementations of attribute grammars, we also consider *remote dependencies* with other nodes' semantic attributes. For this purpose, we equip this class with two operations that are useful in establishing such dependencies. Remote dependencies naturally arise with nodes for elements with a unique identifier. Indeed, we allow the recovery of the semantic nodes associated with such elements (getNodeById method). We also allow the recovery of the semantic node for any other element (nodeFor method). This is useful when dealing with other kinds of remote dependencies (for instance, with attributes for the root). Finally, the SemanticNode class also includes an operation to consult the semantic attributes' value (valueOf method) and an operation to register those semantic attributes (addAttribute method). The Attribute class represents the semantic attributes. To instantiate this class, we must provide a semantic function, which is characterized by SemanticFunction interface. The SemanticFunctionImpl implementation is for use in the SemanticFactoryImpl class.

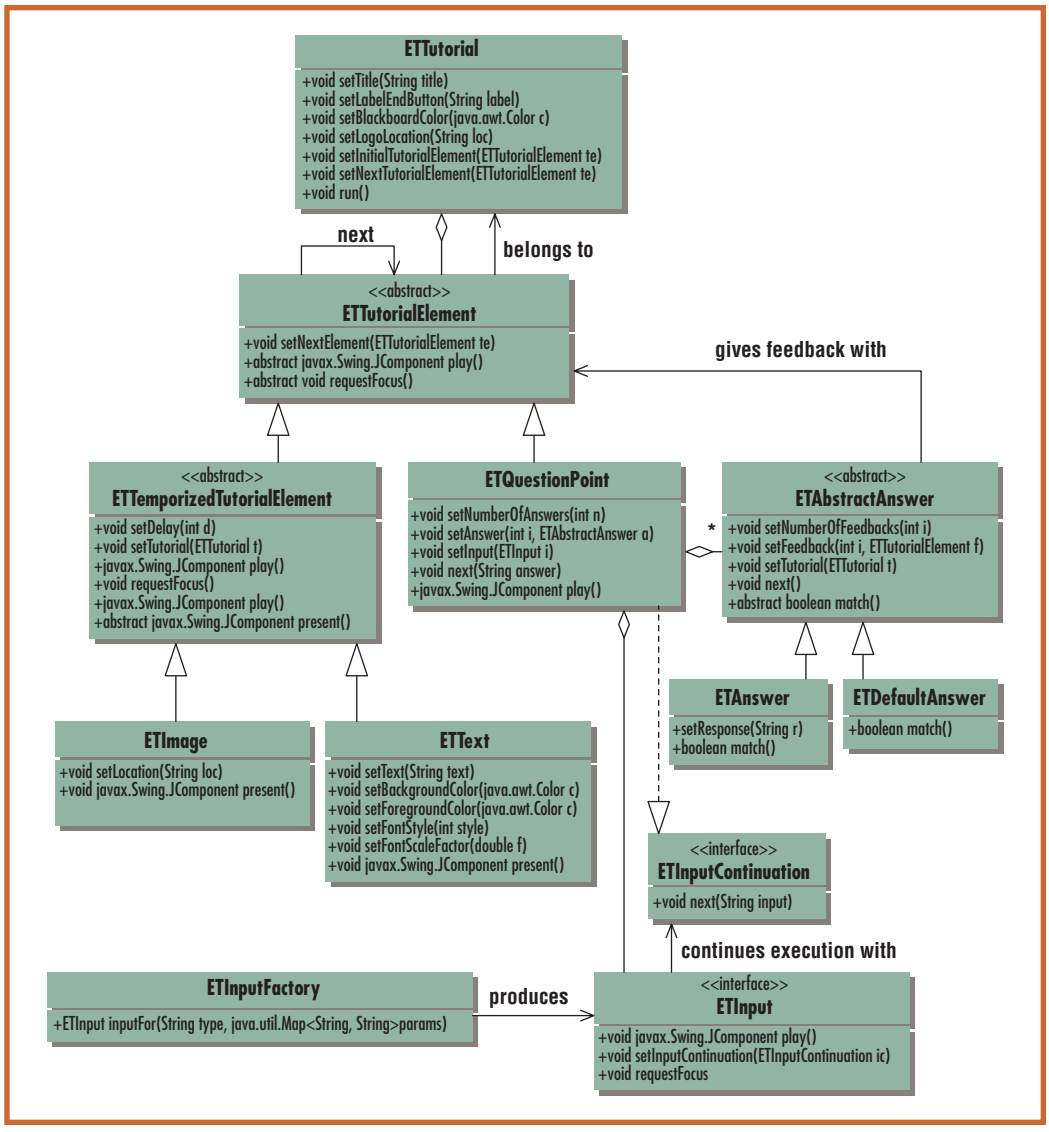


**Figure 4. Main components of the application framework in <e-tutor>.**

## An example

Let's consider the construction of the <e-tutor> application kernel. As for any other kernel, we must provide an appropriate application framework. We also need to implement an application generator that can translate <e-tutor> documents into suitable instantiations of this framework. To do so, developers start with a higher-level design based on the main concepts behind the attribute grammar paradigm; then they implement this design as a semantic module.

### The application framework

Figure 4 outlines the application framework for this kernel. We represent the <e-tutor> structure's main elements as classes, using the Java Swing API. The ETTutorial class stands for the entry point into tutoring systems in general. The ETTutorialElement class is the base for the various elements that tutoring systems can encompass. The classes ETImage and ETText represent the basic speech acts. They are temporized tutorial elements: playing them implies visualizing them and waiting some time before moving to the next tutorial element. The base class ETTemporizedTutorialElement takes care of this common behavior. The ETQuestionPoint class deals with testing learners' knowledge. The class ETAbstractAnswer abstracts the common behavior for the answers: incrementing the associated counter and setting the first speech act of the corresponding feedback as the active one in the tutorial. The ETAnswer and the ETDefaultAnswer classes respectively represent conventional and by-default answers. We also introduce basic interfaces for providing input methods (ETInput and ETInputContinuation). Finally, ETInputFactory lets developers build input methods from an input method's type and a set of configuration parameters.

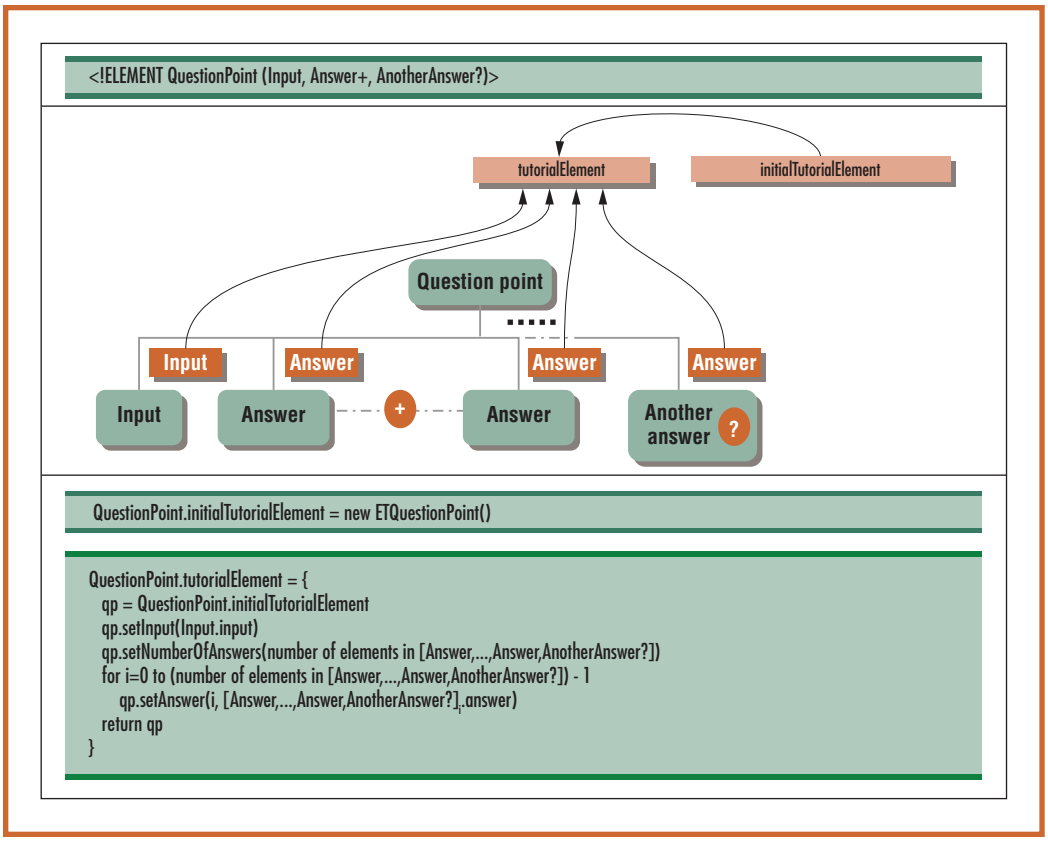


**Figure 5. A design document fragment in the <e-tutor> generator.**

In our <e-tutor> environment, we designed the different components to be easily customizable; we included a *set* method for each relevant feature. We also designed the framework to facilitate its extension for each particular application scenario. Indeed, developers of tutoring systems typically provide input methods to meet instructors' specific requirements.

## The generator

One way to design a generator is to describe in a document how to associate semantic attributes with the document trees' nodes and how to compute these attributes' values. This document would be made up of several entries, each containing

- a fragment of the language's grammar,
- a set of tree patterns representing this grammar fragment and incorporating the semantic attributes, and
- the semantic functions to use in computing each semantic attribute.

Figure 5 outlines one of these entries for the <e-tutor> generator's design document:

- The top line in the figure shows the fragment of the language's grammar describing the syntax structure of the relevant element (QuestionPoint in this example).
- The middle line shows the corresponding tree pattern and the associated semantic attributes for the previous grammar's fragment. When an attribute B depends on an attribute A, we draw an arrow starting at A and finishing at B. Therefore, arrows in the resulting dependency graph suggest information flow during the computation of the semantic attributes.
- At the bottom of the figure and for each attribute with an incoming arrow, we outline the semantic equations by indicating the relevant semantic attributes and the pseudocode of the semantic functions that calculate these attributes' values.

We can easily encode this design using our generation framework. Each equation yields a method in the resulting semantic module (see figure 6).

The document-oriented approach complements conventional software development approaches; it doesn't substitute for them. A key aspect of the approach is to find a good balance between the expressivity of markup languages and their usability by experts. We also assume that

```
public class ETutorSM extends BaseSM {
   public static void setInputFactory(ETInputFactory inputFactory) {...}
   public ETutorSM(SemanticNode sn) {super(sn);}
   @ForAttribute(elements={"Tutorial"},attribute="initialTutorial")
      public Object initialTutorialOfTutorial() {...}
   @ForAttribute(elements={"Tutorial"},attribute="tutorial")
      public Object tutorialOfTutorial() {...}
   ...
   @ForAttribute(elements={"QuestionPoint"},attribute="tutorialElement")
      public Object tutorialElementOfQuestionPoint() {
         ETQuestionPoint qp = (ETQuestionPoint)valueOf("initialTutorialElement");
         qp.setInput((ETInput)children()[0].valueOf("input"));
         qp.setNumberOfAnswers(children().length-1);
         for(int i=1; i < children().length; i++)
            qp.setAnswer(i-1,(ETAbstractAnswer)children()[i].valueOf("answer"));
         return qp;
      }
...
}
```

**Figure 6. Implementation of the <e-tutor> generator's semantic module (excerpt).**

the approach would have high costs for setting up the initial production environment. The development team would need to design suitable markup languages and provide application kernels. Nevertheless, the production team would rapidly amortize this cost with successive production and maintenance iterations. Besides, the effort can also pay off with the development of similar new applications. Finally, the smart use of a generation framework can decrease overall implementation effort.


## Acknowledgments

Spain's Department of Education and Science supported this work through the OdA Virtual project grant no. TIN2005-08788-C04-01 and the AdaptaLearn project grant no. TIN2007-68125-C02-01. The Santander/UCM grant no. 2007/1725 also supported part of the development.


## References


1. J.L. Sierra, A. Fernández-Valmayor, and B. Fernández-Manjón, "A Document-Oriented Paradigm for the Construction of Content-Intensive Applications," *Computer J.*, vol. 49, no. 5, 2006, pp. 562–584.
2. M. Mernik, J. Heering, and A.M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, 2005, pp. 316–344.
3. A. Bork, *Personal Computers for Education*, Harper & Row, 1985.




## About the Authors


**José Luis Sierra** is an associate professor in the Department of Software Engineering and Artificial Intelligence at Complutense University of Madrid and a member of e-UCM, the e-learning research group at UCM (www.e-ucm.es). His research interests include e-learning technologies, domain-specific languages, and markup languages. He received his PhD in computer science from UCM. Contact him at Fac. Informática, UCM, C/ Profesor José García Santesmases s/n, 28040 Madrid, Spain; jlsierra@fdi.ucm.es.



**Alfredo Fernández-Valmayor** is an associate professor in the Department of Software Engineering and Artificial Intelligence, the scientific head of the Virtual Campus, and a coleader of the e-learning research group, all at Complutense University of Madrid. His research focuses on the educational uses of markup languages and the development and authoring of educational materials for Web-based educational systems. He received his PhD in physics from the UCM. Contact him at Fac. Informática, UCM, C/ Profesor José García Santesmases s/n, 28040 Madrid, Spain; valmayor@fdi.ucm.es.



**Baltasar Fernández-Manjón** is an associate professor in the Department of Software Engineering and Artificial Intelligence, the vice dean of research and foreign relationships at the Computer Science School, and a coleader of the e-learning research group, all at Complutense University of Madrid. His research interests are e-learning technologies, educational uses of markup technologies, application of educational standards, and user modeling. He received his PhD in physics from UCM. Contact him at Fac. Informática, UCM, C/ Profesor José García Santesmases s/n, 28040 Madrid, Spain; balta@fdi.ucm.es.