ITS Ontology Engineering:
Borrowing from Design Patterns

Vladan Devedžić
University of Belgrade, FON - School of Business Administration, Department of Information Systems
POB 52, Jove Ilića 154, 11000 Belgrade, Yugoslavia
Phone: +381-11-3971440, +381-11-493492 Fax: +381-11-461221
Email: devedzic@galeb.etf.bg.ac.yu, devedzic@fon.fon.bg.ac.yu
URL: http://galeb.etf.bg.ac.yu/~devedzic/

Abstract
This paper describes a method of ITS ontology development using design patterns. Design patterns are solutions to stereotypical problems in object-oriented software design. The paper briefly overviews the concept of design patterns and shows some experience of using design patterns in ITS design. It also scans some of the most relevant efforts in the field of ITS ontology engineering, and illustrates our own approach to the ITS ontology development, called GET-BITS. Then it elaborates on the relation between design patterns and ontologies, thus building a rationale for the research presented in the central part of the text - development of a concrete ontology, called the Generator ontology, using design patterns. This central part of the paper first focuses on the informal steps that are necessary when developing ITS-related ontologies using the GET-BITS approach. The steps are then illustrated in detail describing the Generator ontology. Finally, the paper discusses other important aspects of the Generator ontology and some new research problems that have been open during its development.

Topic/Subtopic from the proposed list:
• Methods of ontology development/evolution/maintenance/evaluation

Keywords
Ontologies, design patterns, intelligent tutoring systems, architecture, design, application.
Introduction - setting up the context

In developing ontologies for intelligent tutoring systems (ITSs), it helps to pay more attention to the achievements of people from other disciplines. Some people may use different terminology, but they may actually mean much the same as we do. To an extent, people from the design patterns community are like that. So, the following question naturally arises: can we borrow from them?

The purpose of this paper is to show how the technology of design patterns can be applied in ITS ontology engineering. The field of design patterns is actually a subfield of software engineering. Using design patterns along with the other tools of ITS ontology engineering can open new perspectives and shed another light to this growing discipline of AIED research. Specifically, our key objectives and ideas are to show how the concepts of ontologies and design patterns partially overlap, and to illustrate that overlapping by a concrete example of using design patterns in ITS ontology engineering. Note, however, that it is possible to extend the same ideas to the ontology engineering of other software systems.

There is at least a dozen definitions of ontologies. One of the most recent ones says that an ontology provides the basic structure or armature around which a knowledge base can be built [Swartout and Tate, 1999]. We adopt that view of ontologies in this paper. We also support B. Chandrasekaran's observations that "AI is getting more and more integrated with other software disciplines", and that "ontologies are not just for knowledge based systems, but for all software systems - all software needs models of the world, hence can make use of ontologies at design time" (from his invited lecture at the 4th World Conference on Expert Systems, Mexico City, 1998).

Background and previous work - design patterns and ontologies in ITSs

Design patterns and ontologies are two general disciplines, applicable to almost all kinds of software systems, both intelligent and traditional. The interest of AIED researchers for both of these disciplines grows steady.

Design patterns

Design patterns are "simple and elegant solutions to specific problems in object-oriented software design" [Gamma et al., 1995]. They capture static and dynamic structure of these solutions in a consistent and easily applied form. They show generalized, reusable, domain-independent solutions of stereotypical problems that can be used many times without ever doing it the same way twice. It is possible to use design patterns in object-oriented software development in any application domain. For easier reference, each pattern has a descriptive short name that describes the essence of the stereotypical problem that it solves in design.

Design patterns are not invented. They are discovered from experience in building practical systems. There are catalogues of design patterns, in which all of the patterns are described using some prespecified template. For example, the template described in [Gamma et al., 1995] suggests describing each pattern by showing its name, classification and intent, structure, motivation for its use, commenting its applicability and the positive and negative consequences of using it, and discussing its implementation and known uses.

Using design patterns in practice means first considering what family of patterns from the catalogues is related to the particular design problem. There are creational patterns, behavioral patterns, structural patterns, organizational patterns, analysis patterns, etc. [Fowler, 1997]. After finding such a family, the designer considers in what way the patterns from that family solve design problems, what are their intents, what are the consequences of using them, how they are interrelated, and how they increase reusability. Finally, once the right pattern is selected, its description should be read thoroughly, paying particular attention to its structure, participants, and collaborations sections. Then the pattern should be adapted to the particular design problem.

Using the Builder pattern in ITS design

As an example of how design patterns can be useful in designing ITSs, this section describes the case of using the pattern named Builder. The pattern itself is described in many catalogues, using different templates. Its description here is incomplete, and that's on purpose - the idea is only to get a feeling of it from the ITS perspective.

Classification and intent. Builder is a creational pattern. Its intent is to help separate the construction of a complex object from its representation. Such a separation makes it possible to create different representations by the same construction process.

Motivation. Suppose an ITS designer wants to develop an explanation generator that can generate explanations for different students. In general, current level of mastering the subject of the ITS is different for different students at any given moment. That fact is reflected in the student model of each student. Novice students should get more general and easy explanations, while the ITS should generate more complex and detailed explanations to more advanced students. The problem is that the number of possible explanations of the same topic or process is open-ended. It should be anticipated that during the system maintenance another
set of knowledge levels could be introduced in order to describe the student model more accurately. The explanation generator should not be modified each time another set of knowledge levels is introduced. On the contrary, it should be easy to add a new knowledge level easily. Using the Builder pattern provides a solution: the explanation generator can be configured with an ExplanationBuilder, an object that converts a specific knowledge level from the student model to an appropriate type of explanation. Figure 1 illustrates this idea. Whenever the student requires an explanation, the explanation generator passes the request to the ExplanationBuilder object according to the student's knowledge level. Specialized explanation builders, like EasyExplanationBuilder or AdvancedExplanationBuilder, are responsible for carrying out the request. Note that their concrete implementations of the functions like CreateText and CreateGraphics provide polymorphism in generating explanations.

![Figure 1. Using the Builder pattern in designing explanation generator](image)

Structure. Figure 2 shows the general structure of the Builder pattern.

![Figure 2. Structure of the Builder pattern](image)

Participants and collaborations. The abstract class Builder (e.g., ExplanationBuilder in Figure 1) provides interface for creating parts of the Product object (e.g., text and graphics of the explanation). ConcreteBuilder implements that interface. ITS designer configures the Director object (explanation generator) with the desired Builder object (types of explanation, according to the student's knowledge level). Director notifies the builder each time a part of the product should be built. The builder handles the requests from the director and adds parts to the product.

Applicability. The Builder pattern is useful whenever the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled. It is also useful when different representations are needed for the object that's constructed.

Consequences. Using the Builder pattern lets designers vary a product's internal representation (e.g., the contents of the explanation). The pattern provides isolation of the code for representation from the code for construction. Construction of the product is a step-by-step process, and is under the director's control.

Known uses. Examples of using the Builder pattern in ITS design include different generators, such as explanation generator, exercise generator, and hint generator.

Ontologies in ITSs

Although several ITS authors mention ontologies in their work, the most in-depth work on ITS ontologies so far has been done by Riichiro Mizoguchi and his associates ([Chen et al., 1998], [Ikeda et al., 1997].
[Mizoguchi et al., 1996] and Tom Murray ([Murray, 1996], [Murray, 1997], [Murray, 1998]). Their work has been an important source of ideas for our own work on ITS ontology design.

The relevant work in the Mizoguchi Lab is focused on task ontology of ITSs. Task ontology is used as a theory of vocabulary/concepts that create the basis for the development process of knowledge-based systems. It allows users to model the task explicitly, in different abstract levels. The primitives it introduces can be reused by other tasks. The ultimate goal of task ontology research is a theory of the entire vocabulary necessary for building a model of human problem solving processes. Starting from task ontology, several ITS ontologies can be defined, such as domain ontology, student model ontology, teaching strategy ontology, and interface ontology [Chen et al., 1998].

Murray's work on Eon tools for designing content, instructional strategy, student model, and interface of ITSs has produced an elaborated vocabulary, divided into several categories. The categories include topic links/relationships, pedagogical topic properties, student model parameters, teaching styles/methods, question and answer types, pedagogical domain knowledge and instructional objects, knowledge types, primitive tutorial actions, and task types [Murray, 1996]. The key concept in Eon tools is the Ontology object, which defines the conceptual vocabulary and underlying structure for the domain knowledge for each special purpose ITS shell (or for a particular ITS). For example, since Eon tools use a semantic net representation of the tutor's knowledge called the "Topic Network", the Ontology defines the types of topics and topic links allowed in the network.

An important ontology-related concept common to both Mizoguchi et al. and Murray is the concept of levels (layers) in ontology design. Task ontology is divided into three layers: core task ontology, task-specific ontology and task-domain ontology. Core task ontology is a general problem-solving ontology, and it does not depend on any task or domain. It is the basis of the other two layers, since it defines concepts needed for modeling all types of problem solving. Task-specific ontology depends on a certain kind of task type, such as scheduling or training, but it does not depend on any domain. Task-domain ontology describes domain models from the task-type perspective.

In Eon tools, the Ontology defines topic levels, which allow for distinguishing multiple levels of performance (e.g. memorizing vs. using knowledge), mastery (novice to expert), and pedagogical purpose (summary, motivation, example, evaluation, etc.) for each topic. Also, presentations of topics are associated with the knowledge encoded in the topics via topic levels.

The GET-BITS approach to the development of ITS ontology

Our view of ITS ontology comes from our work on development of a model and a framework for ITS design, called GET-BITS (GEneric Tools for Building ITSs). The most complete description of GET-BITS to date can be found in [Devedžić et al., 1999]. The heart of the ITS ontology in GET-BITS is a layered, hierarchical scheme of composition of constituent parts - components and agents - of any ITS, shown in Figure 3. All agents and components in an ITS are defined at one of five levels of abstraction, and along several dimensions (such as knowledge representation, methods, and inference techniques). Primitives are components like plain text, logical expressions, attributes and numerical values. They are used to compose units like rules, frames, and different utility functions. These are then used as parts of certain building blocks that exist in every ITS, e.g. topics, lessons and teaching strategies. At the system level, we have self-contained systems or agents like explanation planners, student modeling agents, and learning actors, all composed using different building blocks. Finally, at the integration level there are collaborative learning systems, distributed learning environments, and Web-based tutoring systems.

<table>
<thead>
<tr>
<th>Level of abstraction</th>
<th>Objective</th>
<th>Semantics</th>
<th>Level of abstraction</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>Integration</td>
<td>Multiple agents or systems</td>
<td>Level 1</td>
<td>D_1</td>
</tr>
<tr>
<td>Level 2</td>
<td>System</td>
<td>Single agent or system</td>
<td>Level 2</td>
<td>D_2</td>
</tr>
<tr>
<td>Level 3</td>
<td>Blocks</td>
<td>System building blocks</td>
<td>Level 3</td>
<td>D_3</td>
</tr>
<tr>
<td>Level 4</td>
<td>Units</td>
<td>Units of blocks</td>
<td>Level 4</td>
<td>D_4</td>
</tr>
<tr>
<td>Level 5</td>
<td>Primitives</td>
<td>Parts of units</td>
<td>Level 5</td>
<td>D_5</td>
</tr>
</tbody>
</table>

Figure 3. Layers and dimensions of the ITS ontology in the GET-BITS approach

The integrated concept of the ITS ontology is split in GET-BITS into a number of separate but interrelated ontologies for all constituent parts of an ITS, such as System ontology, Domain knowledge ontology, Pedagogical knowledge ontology, and Interface ontology). These ontologies are defined in small composable
modules at each level of abstraction and using the top-down approach, so that the knowledge they need can be assembled. All ontologies have the same starting point (the root concept) defined as an abstract knowledge element. In this way, each ontology assigned to a higher level of abstraction includes the taxonomies of the lower levels, and forms a set of inclusion lattices of ontologies.

What is the relation between ontologies and design patterns?
To answer this question, consider first what some of the most prominent representatives of the design patterns community say about the essence of design patterns [Gamma et al., 1993]:

*Design patterns provide a common vocabulary for designers to communicate, document, and explore design alternatives. They reduce system complexity by naming and defining abstractions that are above classes and instances… Design patterns constitute a reusable base of experience for building reusable software. They distill and provide a means to reuse the design knowledge gained by experienced practitioners. Design patterns act as building blocks for constructing more complex designs; they can be considered micro-architectures that contribute to overall system architecture.*

On the other hand, let's see what Chandrasekaran et al. say about ontologies [Chandrasekaran et al., 1999]:

*Ontologies… largely come to mean one of two related things: A representation vocabulary, typically specialized to some domain or subject matter. More precisely, it is not the vocabulary as such that qualifies as an ontology, but the conceptualizations that the terms in the vocabulary are intended to capture… Occasionally, a body of knowledge describing some domain, typically a common sense knowledge domain, using such a representation vocabulary. For example, CYC… (see [Lenat, 1995]) …often refers to its knowledge representation of some area of knowledge as its ontology.*

Summing up these quotations, Swartout and Tate's understanding of ontologies, and numerous other papers and articles both on design patterns and on ontologies, we note several similarities between the two concepts/fields. First, both design patterns and ontologies are about vocabularies, about knowledge, and about "architectural armatures". Both concepts also describe things at the knowledge level [Newell, 1982].

Ontologies are more common sense-oriented, design patterns are more concrete. Design patterns are often about Earthly things such as software design, but can be about more abstract activities as well (e.g., organizational patterns and analysis patterns [Fowler, 1997]). Next, it is possible to draw an analogy between libraries of ontologies and catalogues of design patterns. Although catalogues of design patterns are not ready-to-use building blocks such as ontologies from the libraries, there are some activities already in making them ready-to-use blocks [Coplien and Schmidt, 1995]. Also, it doesn't take a hard mental shift to view ontologies as abstract patterns, or knowledge skeletons of some domains. Likewise, it is not too hard to understand design pattern templates as knowledge of how ontologies of design patterns may look like.

In short, although design patterns and ontologies are not the same, they overlap to an extent. This gives us an opportunity to use design patterns along the other sources of ontology design. Design patterns can at least give many ideas of how to develop ontologies. When used together, ontologies and design patterns can help establish links between the knowledge level and actual design and implementation effectively.

Applying design patterns to ITS ontology design - the Generator ontology
We have used design patterns in GET-BITS as sources of several smaller-scale ontologies that we are developing, thus making our own steps towards design of the overall ITS ontology. It is important to note that we never consider the process of ontology development as "case closed", no matter how elaborated the ontology is. The ontologies we have developed so far will definitely need further refinement. However, we have already had some practical experience with them.

ITS ontology design using design patterns - what are the steps and goals?
Our approach in designing every smaller-scale ITS ontologies has not been strict and formalized. Instead, we can just say that, roughly, the approach is based on the following steps:

- **Start with a least-commitment common underlying structure of several typical processes taking place in almost every ITS.** By "structure" ("armature", to put it in Swartout & Tate's way) we mean both knowledge structure and problem-solving structure.
- **See whether the structure overlaps with any of the known and catalogued design patterns, and to what extent.** Gamma et al. describe how to select appropriate design patterns from catalogues [Gamma et al., 1995]. In case that some patterns considerably overlap, select them as candidates for further consideration. We have not established any strict threshold of considerable overlap; it has rather been a matter of intuition and experience.
- Define a common vocabulary that is consistent with all such processes. Good starting points for defining such a vocabulary can be [Mizoguchi et al., 1996] and [Murray, 1997].
- See how the terms from the vocabulary relate to the terminology of the selected design patterns and possibly make some modifications and generalizations in the vocabulary.
- Design a common ontology, based on the underlying structure of the ITS processes of interest, the descriptions of the selected design patterns, the common vocabulary defined, and any other ITS-specific issue as necessary.
- Starting from the common ontology, design level-specific ontologies where necessary, according to the GET-BITS levels of abstraction. In other words, the common ontology may have some level-specific variants that should be made explicit in the design.
- Make explicit the links between ontologies and other knowledge elements across the GET-BITS levels of abstraction, thus possibly showing multiple-level ontologies.
- Based on the common ontology and the corresponding level-specific and multiple-level ontologies, develop ITS-task-ontologies that fit the needs of specific classes of ITS tasks of different complexity (such as student modeling, interrupting, and providing feedback to the student).

The main common goal of going through these steps is to have at least a vague representational and design framework for developing ITS ontologies. Another goal is to make possible to reuse the knowledge of design patterns in multiple ways. The following subsections show how a concrete ontology - the Generator ontology - has been designed using this approach.

Generators in intelligent tutoring and learning - some common examples

In a typical session with an ITS, the student can ask for an explanation of a specific topic of interest. In response to such a request, the system has to generate an explanation, based on the domain knowledge, pedagogical structure of the domain, the student's level of mastering the subject of the ITS, the teaching strategy used, and the attributes of the current session. The contents of the explanation can be text, graphics, sound, video, natural language, speech, and also a combination of such elements. An important property of the explanation is its composition, which includes its objectives, order and placement of the individual elements that make up the explanation, and the presentation style. It is the task of the explanation generator to construct, structure, and present the explanation in an appropriate way, taking many things into account. They include the knowledge type of the topic for which the student requires the explanation (i.e., whether it is a procedure, a fact, a concept, and the like), the student's current progress in learning about that topic, the type of explanation the student has asked for (e.g., clarification, interpretation/evaluation, orientation, details), and many more. It is equally important for a good explanation generator to be able to create different kinds of explanation by essentially the same construction process. For example, the student can ask for an introductory clarification of a topic, or for a detailed one. Although the contents of the two explanations can differ a lot, the process of constructing them should not.

As another example of a generator in an ITS, consider interrupting the student's current action. Again, the system has to generate the interrupt and present some appropriate contents to the student. The cause of the interrupt may be, for example, the student's misconception in learning, heading the wrong way, heading the right way (occasional encouragement - congratulation), or delay in response to the system's actions. The interrupt generator must use the domain knowledge, the student's mastery level and current progress, the rules of teaching strategy, and the attributes of the current session in order to decide on the moment of interrupt and generate the contents to show to the student. These contents can vary a lot, depending on the cause of the interrupt, the student model and the teaching strategy. Examples of the contents of interrupts include various suggestions, advises, reminders and acknowledgements, possibly combined with some graphical symbols (e.g., smiling face) or emotional expressions of animated characters representing pedagogical agents. As in the case of an explanation generator, it should be possible for an interrupt generator to create and assemble different contents of the interrupts in more or less the same way.

Other examples of generators in ITSs include hint generators (hints can be structured, i.e. multilevel), example generators, and different feedback generators (e.g., tell consequence/implication of student's action, tell what is relevant or what to ignore, remind of what they know, reframe the problem or goal, remind of the goal, ask for reasons, ask for prediction, ask for certainty/confidence, and ask if answer 'makes sense'; see [Murray, 1997] for a number of other examples of feedback).

The Builder pattern as the basis of the Generator ontology

All of the above examples have a lot of it in common. Essentially, it is always a certain generator there, and it typically constructs some composite result. It also always uses some knowledge, and it does it in a certain context. Depending on the context, each generator creates different representations by the same construction
process. The Generator ontology captures the underlying common properties and structure of all such
generators.
Scanning available catalogues of design patterns reveals that the Builder pattern best overlaps with the idea of
the Generator ontology. It follows from Figures 1 and 2 that it is actually the ConcreteBuilder that does the
real work, and that the Builder is just a common interface for all ConcreteBuilders. Since different
ConcreteBuilders generate different results (products), the client that uses the generator should actually create
an appropriate ConcreteBuilder each time a result has to be generated, and use it as one-time agent to do the
job. Equally important, the Generator/Director always applies the same construction process (Construct ())
and the same interface, which considerably simplifies the design and increases reusability.
However, the Builder pattern doesn't reveal some other important characteristics of the generators. For
example, the context of generating results is only implicitly present in the pattern (the Construct and
GetResult methods). Also, the knowledge of how the result (the product) has to be created is hidden
somewhere inside the BuildPart method, while in the ontology it should be made explicit. Hence the Builder
pattern can be used only as the basis of the Generator ontology, not as the ontology itself.

The Generator ontology as a common ontology

Starting from the vocabulary used in different generators and the vocabularies of ITSs proposed in
[Mizoguchi et al., 1996] and [Murray, 1997], as well as from the small vocabulary of the Builder pattern (see
Figure 3), the Generator ontology can be defined at the lexical level as a common ontology shown in Table 1.
The table shows the most important concepts in the ontology and some relations that hold among the top-level
concepts and the lower level ones. For example, the Item generator is a kind-of the upper-level concept, i.e.
the Generator. Likewise, the Student model concept comes from another ontology, but is certainly used-by the
Context concept of the Generator ontology.

Table 1. Partial lexical-level specification of the Generator ontology

<table>
<thead>
<tr>
<th>Concept</th>
<th>Relation to the upper-level concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generator</td>
<td></td>
</tr>
<tr>
<td>Item-Generator</td>
<td>kind-of</td>
</tr>
<tr>
<td>Contents-Generator</td>
<td>kind-of</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Concrete generator</td>
<td></td>
</tr>
<tr>
<td>Client</td>
<td></td>
</tr>
<tr>
<td>Builder</td>
<td></td>
</tr>
<tr>
<td>Result/Product</td>
<td></td>
</tr>
<tr>
<td>Item</td>
<td>kind-of</td>
</tr>
<tr>
<td>Text</td>
<td>kind-of</td>
</tr>
<tr>
<td>Graphics</td>
<td>kind-of</td>
</tr>
<tr>
<td>Video</td>
<td>kind-of</td>
</tr>
<tr>
<td>Sound</td>
<td>kind-of</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Contents</td>
<td>kind-of</td>
</tr>
<tr>
<td>Explanation</td>
<td>kind-of</td>
</tr>
<tr>
<td>Hint</td>
<td>kind-of</td>
</tr>
<tr>
<td>Example</td>
<td>kind-of</td>
</tr>
<tr>
<td>Lesson presentation plan</td>
<td>kind-of</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Context</td>
<td></td>
</tr>
<tr>
<td>Session parameters</td>
<td>used-by</td>
</tr>
<tr>
<td>Student model</td>
<td>used-by</td>
</tr>
<tr>
<td>Constraints</td>
<td>evaluate, compute</td>
</tr>
<tr>
<td>Knowledge</td>
<td></td>
</tr>
<tr>
<td>Teaching material</td>
<td>use-by</td>
</tr>
<tr>
<td>Teaching strategy</td>
<td>use-by</td>
</tr>
</tbody>
</table>

Relations among the top-level concepts of the Generator ontology are represented in Figure 4. The Client first creates the one-time Concrete generator, and then it creates the Generator in order to ”wrap-up” the Concrete
generator for the process of building the Result. In fact, by making the Concrete generator a part of the Generator the Client configures the Generator with a desired Concrete generator. The Builder interface of the Concrete builder, as well as the necessary Context and Knowledge need not be created - they already exist at the moment when the Client wants the Generator to construct some Result. Thus the Generator and the Concrete generator just use them, respectively. An important fact omitted from Figure 4 for clarity is that the Result can be either a Leaf result or a Composite result. This comes from representing the Result with another common ontology, developed using the Composite design pattern (see [Devedžić, 1999] and [Gamma et al., 1995] for details on representing composite products/results).

The methods of the Generator ontology include the following:

- **Generate**
  - **Semantics.** When the Client is finished with the process of creating its collaborators (Concrete generator and Generator), it sends the Generator the request to generate the Result (for all parts of the desired Result: Builder→Build Part).
  - **Input.** Parameters needed to describe the Result that the Client wants, the Knowledge needed in order to generate the Result, and the Context of generating it.
  - **Output.** Invoking the Builder→Build Part.

- **Build part**
  - **Semantics.** The Concrete generator builds parts and adds them one by one to the Result in its own, specific way. Building a part can be a recursive process if the part is a composite one. The Builder's method with the same name is just a common interface of all Concrete generators.
  - **Input.** Parameters needed to describe the part to be built.
  - **Output.** The part.

- **Add part**
  - **Semantics.** Add the part built by the Build part method to the composite Result.
  - **Input.** The part built by the Build part method.
  - **Output.** A partial Result. When the last part is added, the Result is complete.

- **Evaluate**
  - **Semantics.** Compute, calculate, measure, or estimate parameters and constraints from the Context, and see how they affect the process of generating the Result.
  - **Input.** A set of parameters and constraints from the Context.
  - **Output.** A set of parameters that may modify, violate, or disable generating the Result.

- **Consult**
  - **Semantics.** Select and use the relevant Knowledge in order to guide the process of generating the Result.
  - **Input.** A set of pointers to the relevant knowledge.
  - **Output.** None.

- **Get result**
  - **Semantics.** Get the Result when it is completed.
  - **Input.** None.
  - **Output.** None.

**Generator ontologies as level-specific, multiple-level, and ITS-task-ontologies**

It is possible to use the common Generator ontology at several GET-BITS levels. Some examples are shown in Figure 5. It is easy to imagine the student's request for an explanation, a hint, an example, a simulation, etc.
at Level 2. In response, the ITS needs to use its explanation generator, hint generator, or example generator, respectively. What we have in all such cases is a request to a specific "subsystem" of the ITS to generate some contents, most often on the screen, based on the student's action, the context of the session, and the relevant domain and teaching knowledge. That is the essence of the level-specific Contents generator ontology that we have developed in GET-BITS at Level 2. Note, however, that in most cases the result of constructing the contents is a composite Level 3 element - explanations, hints, examples, and simulations are knowledge elements defined at Level 3 in GET-BITS. The context of generating the contents, C-context, reflects what in what way the explanation/hint/example/simulation should be presented, given the current status of the session, the student's current progress, the availability and appropriateness of the type of the contents that the student has required, and the like. The knowledge necessary to generate the contents includes details of the topic for which the contents are required, the level of the explanation/hint/example/simulation (e.g., introductory, advanced, expert, details), the strategy of structuring the contents, the prior knowledge required from the student in order to understand such contents, as well as the system's goal when presenting the contents (e.g., to advise, to help, or to confuse the learner on purpose!).

Figure 5. Linking levels of abstraction through ontologies

We can tell a similar story at Level 3. In order to assemble the contents being composed at Level 2, the Contents generator needs different items. For example, a hint that appears on the screen may contain some text, some appropriate graphics, and a cute animated character to increase the learner's concentration and motivation. An item itself may be a Composite item and may have a recursive structure (e.g., there may be several levels in a hint). Most simple items or Leaf items are Level 4 objects in GET-BITS (text templates, graphics, sound, and video). Several kinds of item generators are defined in GET-BITS at Level 3, and what they have in common is the knowledge encoded in the Item generator ontology. This level-specific ontology is also a special case of the common Generator ontology. This time, the "client" is the request for an item that comes from the contents generator operating at Level 3. Each Concrete contents generator creates a series of concrete requests for items. Any such a request is sufficient for creating an instance of a Concrete item generator and an encompassing Item generator. The knowledge that the Concrete item generator uses for generating the item, I-knowledge, is mostly encoded in the descriptions of topics and objectives of the current lesson. For example, a topic object usually contains links to templates for constructing explanations. The Concrete item generator also evaluates constraints described in the I-context. An example of such a constraint may be "student's mastery level is higher than 3 (on a certain scale)". It may be necessary to evaluate such a constraint because the relevant I-knowledge may specify that graphical items of hints are unnecessary if the student's mastery level is higher than 3.

It is also possible to interpret the entire Figure 5 as a partial representation of the multiple-level ontology of Contents generator. Multiple-level ontologies in GET-BITS span across two or more levels of abstraction,
explicitly showing the links between the participating level-specific ontologies and other knowledge elements. The multiple-level ontology of Contents generator includes several level-specific ontologies (including the partially shown Item ontology at Level 4). The important relations between the participating level-specific ontologies are the create link between the Concrete contents generator and the Request for item and the part-of relation between the Item and the Contents.

Finally, starting from the multiple-level ontology of Contents generator, we have partially defined three ITS-task-ontologies so far: the Explanation generator ontology, the Hint generator ontology, and the Lesson presentation planner ontology. We have also started some efforts towards defining the Example generator ontology. The process of the ITS-task-ontology definition is essentially the process of further specification of parts of the corresponding level-specific or multiple-level ontology. It is conducted according to a specific ITS task, such as generating explanations. For example, in the case of the Explanation generator ontology, we had to define the Contents as Explanation (having parts like Highlights, Pointers, Presentation mode, and the like), the Knowledge has become Explanatory knowledge, and some of the Concrete explanation generators are depicted in Figure 2. We believe that such ITS-task-ontologies are the main points of interest of ITS designers, while the other kinds of ontologies are interesting primarily to ontology engineers.

Discussion - what else is important?

There are no ontologies in our approach that are tied to any specific domain of teaching. All of them are ITS-wide ontologies. However, it is possible to draw some analogies between the kinds of ontologies we use and those of the other approaches. The analogies are related to the fact that more specific ontologies are rooted in more general ones, thus actually defining layers of ontologies. In the world of ITSs, the role of our common ontologies is reminiscent of the role core ontologies play in other approaches [Mizoguchi et al., 1996], [Ikeda et al., 1997], [Van Heijst et al., 1997]. However, core ontologies are general and are not specific to ITSs. But if we accept that analogy, then it is possible think of level-specific and multiple-level ontologies of ITSs as being rooted in common ontologies in much the same way task-specific ontologies (described in [Ikeda et al., 1997]) are rooted in core-task ontology. However, these two concepts are not the same. Extending the analogy, it is easy to note that ITS-task-ontologies on a smaller (and domain-independent) scale correspond to task-domain ontologies of Ikeda et al. [Ikeda et al., 1997].

The Generator ontology and its more specific variants are currently implemented using our in-house tool called DON (Designer of ONtologies), which has been developed as a part of the GET-BITS project (see [Devedžić et al., 1999] for details about DON). They have been used in only one practical ITS so far - the FLUTE system that helps students to learn about formal languages and automata [Devedžić and Debenham, 1998]. We would like to stress DON's possibility of customizing the vocabularies used in developing ontologies. The importance of vocabulary customization in ITS ontology development has been also discussed elsewhere (e.g., [Murray, 1998], [Murray, 1999]).

The experience we have had with the FLUTE system has revealed an important ontological commitment related to the layered organization of the Generator ontology: in order to make the Generator ontology really useful, it is necessary to have a layered organization of the domain knowledge and pedagogical knowledge as well. This means that all Topics, Lessons, Teaching Strategies, and other knowledge must be also in multiple levels. Moreover, the same holds for the student model. Other authors have come to similar conclusions (see [Murray, 1996] for an elaborated discussion).

The Generator ontology is also an example of the concept of graph of ontologies, discussed by Van der Vet and Mars [Van der Vet and Mars, 1998]. In such a graph, individual ontologies are connected by directed edges that stand for information-discarding transformations. An individual ontology at some level can be obtained from one or more ontologies at a more detailed level using information-discarding transformations. If the graph is a tree, then every subtree is itself an ontology, and every level of details constitutes a legitimate and correct ontology. Different ontologies and levels of abstraction at Figure 5 directly illustrate all this. We cannot prove the completeness of an ontology [Gomez-Perez, 1996], so in practice we usually limit our focus to the ontology at a certain desired level of details.

Last, but definitely not the least, some comments are necessary regarding knowledge sharing. How can we integrate any of our ontologies with some other ITS ontologies, such as task ontology or Topic ontology? Although in theory that task should be straightforward, practice often shows that there are problems in integrating ontologies developed with different tools. Different vocabularies and representations are not the only ones to blame for that. For example, Valente et al. have reported several problems in trying to automatically translate ontologies from Ontolingua to Loom [Valente et al., 1999]. The problems they have had were due to the translators. A possible solution ontology integration problem can be also borrowed from design patterns: there is the Adapter pattern [Gamma et al., 1995]. It shows how to convert the interface of a class into another interface that clients expect. Thus Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. Chandrasekaran and Josephson suggest a similar solution, although they don't refer to design patterns [Chandrasekaran and Josephson, 1997]. They propose using adapters to help
bridge the ontology gap between a problem and a method, i.e. to map the domain goal to the problem-solving method goal. We believe that the idea of adapters and the Adapter pattern can be a good basis for integrating our ontologies with other ITS ontologies.

Conclusions
The main reason for trying to use design patterns as possible starting points in developing ontologies is the fact that design patterns already do define things and phenomena at the knowledge level. That is exactly what we need for development of ontologies; it is at least a part of what is needed. Moreover, design patterns are very good sources for initial development of ontologies because of their rather informal description. In spite of the fact that design patterns are usually described using a certain template, their description offers a lot of flexibility. They are adaptive to specific design problems. In spite of the fact that patterns often focus on software design problems, they can suggest the structure of ontologies, i.e. some terms, some concepts, and some relations between them. There are also patterns that focus on organization and analysis problems. Such patterns can be useful in developing ontologies as well.

Based on the knowledge represented by some concrete design patterns, we have developed the Generator ontology for ITSs. The ontological analysis of different generators that are used in practical ITSs has clarified the structure of knowledge that we have encoded in our Generator ontology. The development of the Generator ontology and some other smaller scale ontologies useful for ITSs has helped us formulate the steps in the process of developing ontologies using design patterns.

Our work on ITS ontology has opened several new research problems that we plan to tackle in the future. First, further work is needed in order to make possible to integrate our ontologies with those developed by other authors. Ideas like adapters [Chandrasekaran and Josephson, 1997], the Adapter pattern [Gamma et al., 1995], and mappings between ontologies [Schreiber et al., 1994] may help solve that important problem. These ideas have not been explored in the AIED community so far.

Second, with the advance of research and developments of ITS ontologies, the AIED community will need to agree upon the terminology, the ontologies, and the mechanisms used to define new ontologies. This is another area where the experience of the design patterns community can be used: there exist pattern languages. A pattern language is not formal language. It is rather a structured collection of interrelated patterns that provides vocabulary for talking about a particular problem. Pattern languages help designers communicate architectural knowledge, help people learn new design styles, and help new developers avoid traps and pitfalls of unsuccessful design. There is an extremely fertile ground within the AIED community for growing pattern languages, the work that has not started yet (at least not explicitly).

Finally, additional research effort is needed in order to merge the ITS ontology engineering with other subfields of AIED. Standardization of the ITS terminology, which is already under way, should be the first step towards that goal.

References


[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Reading, MA, 1995.


