Software Patterns in ITS Architectures

Vladan Devedzic, *Department of Information Systems and Technologies, FON -School of Business Administration University of Belgrade, POB 52, Jove Ilica 154, 11000 Belgrade, Serbia and Montenegro*
devedzic@fon.bg.ac.yu
http://fon.fon.bg.ac.yu/~devedzic/

Andreas Harrer, *Institut für Informatik und Interaktive Systeme, Universität Duisburg-Essen, Lotharstr. 63, 47057 Duisburg, Germany*
harrer@collide.info
http://www.collide.info/~harrer/

Abstract. The paper discusses ITS architectures in terms of patterns that exist within them. The concept of patterns has received surprisingly little attention so far from researchers in the field of ITS. A recent analysis of a number of existing ITS architectures has revealed that many ITS designers and developers use their own solutions when faced with design problems that are common to different systems, models, and paradigms. However, a closer look into such solutions and their comparison often shows that different solutions and the contexts in which they apply also have much in common, just like the corresponding problems do. In all such cases we can talk of the existence of patterns. A pattern is a generalized solution of a typical problem within a typical context. Discovering such patterns can help clarify general guiding principles of ITS-architectural design in a more systematic way.

Keywords. Intelligent Tutoring Systems, Software Patterns, Software Architecture

INTRODUCTION

What are patterns, speaking architecturally?

The concept of formalising patterns as solutions to typically recurring problems was introduced by the architect Christopher Alexander and his research group (Alexander et al., 1977; 1979). In their work, patterns were applied in architectural design for urban planning, single buildings, and detailed room arrangement. This work was adopted successfully in the area of software system engineering in the 1990s. In software engineering, patterns are attempts to describe successful solutions to common software problems (Schmidt et al., 1996). Software patterns reflect common conceptual structures of these solutions, and can be applied over and over again when analyzing, designing, and developing applications in a particular context. Patterns help designers capture the core structure of the systems they build and provide a disciplined format for sharing ideas that too often remained locked in the heads of a few people.

Descriptions of patterns in a clear literary form provide a vehicle to communicate these ideas; the pattern-oriented mindset of system designers provides a setting in which these ideas can effectively be gathered and shared.

The big impact patterns had in the last years in the field of software engineering can be explained by the fact that patterns give general and reusable solutions to specific problems in software development. Patterns emerge from successful solutions to recurring problems and with the knowledge of patterns it is not necessary for a software developer to solve every software problem himself -the developer can benefit from the experience of other software engineers with similar problems. This is important because software systems tend to grow larger and larger and the need to keep the systems manageable and extensible is therefore growing as well. Since intelligent tutoring systems are software systems with great complexity, it is advisable to transfer general knowledge and trends from the field of software design and architecture, such as that of software patterns, to the field of ITS. It would make it possible to build ITSs with reduced effort, or at least make them easier to maintain in the future.

This is not to say that one can talk of patterns in ITS only in the context of ITS architectures. On the contrary, there are many kinds of patterns in the way learners learn and in the way teachers teach, and all of them can be used as starting points when designing ITSs. For example, there are patterns of speech acts that occur in pedagogical interactions (Katz et al., 1999), patterns in instructional design (Inaba et al., 2001), patterns in teaching strategies (Scott & Reif, 1999), and so on. However, in this paper we do not cover in detail all possible subject areas of ITS where patterns do exist. The section related work gives pointers and relations to various types of patterns relevant for ITSs. In this paper we focus on ITS architectures, i.e. *software* patterns in ITS design. In the context of ITS architectures, we also don't cover all of them, but only selected ones -all of the patterns presented are related to common and well-known ITS concepts, such as collaborative learning, learning companions, and pedagogical agents. This avoids repeating elaborations of general software patterns that have been previously collected in textbooks (such as Gamma et al., 1995) and conference series (such as the Pattern Languages of Programs (PloP) conferences) and concentrates the presentation on ITS- and AIED-relevant aspects.

The reader should understand that there are several different kinds of general software patterns; see (Devedzic, 2002) for an extensive discussion on kinds of software patterns. For the purpose of the following sections, it helps if one can differentiate between *design patterns* - common design structures that are useful in creating reusable software (Gamma et al., 1995), and *analysis patterns* -reusable models resulting from the process of software analysis applied to common business problems and application domains (Fowler, 1997). Both kinds apply to software architectures; the difference exists in terms of perspective and phases of the entire software development process, as well as in terms of levels of abstraction and details. Design patterns are more concrete and relate to composition of usually smaller pieces of software into coherent larger modules when the functionality and behaviour of the larger pieces is specified beforehand. Analysis patterns are related to identifying specific roles and functionalities of software modules within a software system as a whole, and to specifying their typical interactions when they are put together in order to achieve a desired system behaviour. True, this categorization is not a clear-cut one and can be confusing to non-experts in software engineering. The reason is that the border between software analysis and software design is not strict either. Moreover, one should note that *many* artefacts result from the interwoven software analysis and

design processes. The system architecture is just one of such artefacts. It describes how a software system is composed from identifiable software *components* and *connectors* of various distinct types. Patterns for software architectures (or *architectural patterns* (Buschmann et al., 1996)) can be identified by abstracting from the details of software *architectural styles*, which are defined for families of architecturally related software systems (e.g., layered architectures, pipeline architectures, repositories, etc.).

Some patterns described in this paper can be categorized as design patterns, some other as analysis patterns of ITS architectures. The respective sections specify the pattern categories explicitly. Note that there are several widely used ways ("templates") of describing a pattern. One of them to specify the *context* where the pattern is useful, the *problem* that the pattern addresses, the *forces* that drive the process of forming a solution, and the *solution* that resolves those forces (Fowler, 1997). Specifying the solution often involves showing a *diagram*, and sometimes the pattern's *variants* are also described and pointers to *related patterns* are given (Gamma et al., 1995). Most of these "parts" of a pattern are usually shown in the form of simple statements. In this paper, we extended this template by the pattern *category* statement. Using such a template ensures for consistency in describing patterns.

The patterns described in this paper do not imply that they are the only architectural patterns for the application types (e.g. CSCL or pedagogical agents) which can be identified. They only represent good examples of patterns that we have discovered.

MOTIVATION

Our motivation for conducting research described in this paper was three-fold. First, we wanted to provide explicit general guidelines of ITS-architectural design in the context of general software design. The idea has been to stress the engineering aspect of ITS architectures in terms of some simple but explicit rules that should govern ITS-architectural design and can lead to more elegant and more stable architectures. Such architectures exhibit functional clarity and simplicity of the system's modules, are free from unnecessary details at a given level of abstraction, enable a high degree of scalability, ensure the system's robustness for exploitation, need not be modified over a longer period of time during the system maintenance and upgrades, and are largely reusable.

Second, we have found it very stimulating for further research to view well known learning theories, teaching strategies, and interaction metaphors from the system design perspective. For example, if we modify parts of the architecture of a collaborative learning environment, what are the consequences from the learners' perspective? What architectural modifications should we make in order to increase their interest in collaboration? Likewise, how do we design a pedagogical agent in order to make it more convincing as an artificial peer learner? Revealing common patterns in ITS architectures is a way to:

- shed more light onto existing ITS architectures;
- extract and specify key design elements common to many ITS architectures;
- increase the AIED/ITS community's awareness of the forces that drive design of ITSs;
- provide a more systematic approach to the design of new ITSs.

The third guiding force of our research has been a striving to compile some segments of well-established knowledge and experience of ITS designers, abstract common ideas, and present them in the context of already known general software patterns. Although that aspect of our research is the least represented one in *this* paper, earlier extensive reports indicate vast research opportunities in that direction (Devedzic, 2001; 1999a).

WHY USE PATTERNS IN ITS?

The idea of patterns is not new in the AIED/ITS community, but so far it has not been exploited very much. On the other hand, there's a lot of reasons why it should be. The knowledge of patterns can provide simple and elegant solutions to specific problems in ITS design. It can also help in improving the organization of the vocabulary of different learning and teaching styles and strategies. Patterns have a great potential in generating and structuring explanations, hints, simulation, and other feedback that students require from an ITS. They could represent the cores of solutions to analysis, design, architectural, instructional and other problems in ITSs, the solutions that have been used more than once in different systems. Patterns can capture both the static and dynamic structure of these solutions in a consistent and easily applied form.

The patterns described in this paper are focused primarily on architectural and structural issues of ITSs. Therefore, the main use of such patterns lies in their potential to improve the design of ITSs and to make the development, maintenance, and extension of these systems easier. The complexity of ITSs makes their development a hard task, hence the use of know-how from software engineering within ITSs and the explicit elaboration of the know-how implicitly contained in existing ITSs becomes increasingly important.

What exactly is the benefit of patterns? Why is this work important and timely? Who will benefit from patterns? "If you build them, they will come" -is it necessarily so?

To answer these questions, recall that although up to now there *were* some successful ITSs, they are usually not yet *widespread* in schools, training centres, at universities, and in other educational institutions (with the exception of a few broadly disseminated systems, such as the AlgebraTutor (Koedinger et al., 1997) or the SQL tutor (Mitrovic, 2003)). If we want ITSs to get really widely used in practice, it is necessary to satisfy a number of conditions. Some of them are that developers build more and more competitive and useful systems, that at least some of them get accepted and are really popular with different communities of learners, and that their overall quality keeps constantly improving. A common denominator of all these conditions is that developers design, implement, thoroughly test, and periodically evaluate ITSs not only from pedagogical and educational perspectives, but also following sound rules of software design. True, one of the most important objectives of developing ITSs is to improve the learning efficiency in specific domains, and many people might not worry much about architectures and design of ITSs. However, they are still a kind of *system*, and somebody still has to build them. Moreover, being *software* systems that help learners, teachers, and authors actively participate in different educational tasks and processes, ITSs should certainly include some advanced *software-design* know-how as an engineering support for such pedagogical goals and instructional processes.

Software patterns are tools that embody parts of software-design know-how. There is a huge evidence of successful application of patterns in software engineering in different application

domains; see (Buschmann et al., 1996; Gamma et al., 1995; Fowler, 1997; Schmidt et al., 2000), as well as http://hillside.net/patterns/patterns.html and http://st-www.cs.uiuc.edu/users/patterns/patterns.html. If applied in building ITSs, patterns can "provide a more systematic approach to design of new ITS" in much the same way that they did in building various business information systems, medical systems, or manufacturing systems. It is important, however, to understand that some software patterns are domain-neutral, whereas others are domain-specific. This paper focuses on domain-specific ones, i.e. on some ITS-specific patterns.

The AIED/ITS community can benefit from patterns in a number of ways. For example, being aware of the existence of some specific patterns and with a good understanding of what good they are, developers can reuse architectures of successful ITSs when building other similar ITSs. Entire modules of such architectures and even the ways they interact with other modules can be shared as components if they implement functionalities that are useful in other systems as well (such as a database of learners, test scores, and structuring of references for further reading, to name but a few).

PATTERN DISCOVERY

"If patterns are really such nice things, where do I find them?"

Let's make ourselves clear from the very beginning: nobody should *invent* patterns. They are rather *discovered from experience* in building practical systems. Hence the answer to the above question is either a) ready-to-use software patterns exist only in pattern catalogues and repositories, where other researchers, designers, and developers have described and categorized them; or b) "Go discover them yourself!"

Note, however, that although b) is challenging, it is far away from being easy and straightforward. In the beginning, it is usually a matter of intuition and background knowledge. In our case, we just had a feeling in the beginning that different ITS architectures hide several common principles and solutions. What followed was an extensive analysis of organization and architectures of existing ITSs and their components in search of some possible common design decisions, common interactions among components, and common generalized principles underlying superficially different designs. More precisely, we were trying to *extract* patterns from numerous known examples, systems, architectures, designs, learning and teaching styles, strategies, etc. That takes a lot of time and effort, and there are always exceptions to the patterns that are discovered eventually.

"Is this a pattern or not?"

Suppose we have identified a piece of design that recurs regularly in ITS architectures. Winn and Calder suggest how to determine whether that piece is a pattern or not (Winn & Calder, 2002). First of all, all patterns are generative -they have a number of concrete (and slightly different) instances. Also, a pattern is "both a process and a thing", i.e. both a description of a solution of a common problem in a given context, and a description of the process which will generate that solution. It implies an artefact, i.e. it describes how the software works and the relationships it tries to capture. A pattern should express possibility (what decisions could be (or were) made in a particular context) and feasibility (desirable decisions in a particular context, or the reasons why they have been made). It must stress both invariant parts of the solution to a recurring problem, as well as those parts of a solution likely to change as a developed system

evolves (both stable and changing system elements). A pattern is grounded in a domain - discussion of a pattern has no meaning outside the domain to which it applies. Finally, each pattern is part of a *pattern language* -a collection of patterns that "captures" a domain by identifying its key concepts and their relationships.

There are a number of common pitfalls and difficulties in discovering patterns; they all exist in discovering ITS patterns as well. Perhaps the most common one is an attempt to discover something too concrete or too abstract. It is important to understand that a pattern bridges many levels of abstraction -it is neither just a concrete, designed artefact nor just an abstract description. Also, trying to identify whether some just discovered pattern has been used in an existing system should not bring difficulties -if it has, the pattern will be clearly present and recognizable. Moreover, a piece of design is not a pattern if it fails to pass the "frequency of use" criterion -the pattern's existence lies in its recurring, identifiable presence in artefacts. Minimum requirement for a pattern is that it has to be applied successfully more than once (Gamma et al., 1995) in different systems (preferably of different creators/designers). Other sources give "the rule of three", i.e. the minimum of 3 occurrences in real systems, to qualify as a pattern. In our own research, we were constantly trying to see how strong the support is for the patterns we have discovered in existing ITSs and in on-going projects (how many different architectures use the pattern, in what part, on what purpose, and the like; see the next section). Note, however, that there is no strict threshold for the frequency of use -it largely depends on the domain itself and on the number of systems in that domain that have been developed. It is obvious that the number of systems considered as sufficient evidence that a certain pattern is used must be larger in, say, the domain of business information systems than in the domain of ITS. Not even the relative numbers (i.e., the percentage) are of much help, because domains are inherently different and also the evidence usually changes as new systems are developed. To prove the qualification as a pattern, experts in software patterns recommend accurately maintaining and referencing each piece of evidence/recurrence of discovered patterns.

Last but not least -patterns are not about solutions to trivial problems, so not every solution to a software design problem warrants a pattern (Winn & Calder, 2002). This is not to say that all patterns necessarily have complex structures -on the contrary, there are software patterns that have a very simple structure (Fowler, 1997).

Our experience shows that most ITS-architectural patterns focus on key aspects of software design of these systems, aspects that designers in that area face time after time, again and again, in one form or another. These aspects, in turn, stem from continuous efforts to architecturally support important instructional design and cognitive issues.

METHODOLOGY

In the course of discovering the patterns described below, we have analyzed different ITS architectures that have been described in a number of AIED- and ITS-relevant journals and conference and workshop proceedings since 1997. We also consulted a number of designers and developers of specific systems, asking them for clarifications and detailed documentation of the architectures they developed (thus our elaboration is more detailed for the time period from 1997-2002, because additional information could be gained in direct exchange with the developers for a longer period; nevertheless we found evidence of characteristic concepts and

systems in recent work for each presented pattern, too). We are perfectly aware of the fact that we might have missed some other important work on ITS architectures, especially since only a few articles in this community directly address architectural aspects.

However, we have tried to maintain some simple statistics related to the patterns we have discovered and the sources we have used. For all the patterns we have discovered, we counted support (how many different architectures use the pattern, in what part, on what purpose, and the like). We have excluded the systems and the papers that we ourselves have authored/co-authored, due to the possibility of some bias in the analysis. We have also excluded the systems if they were architecturally much akin the earlier versions of the same systems, as well as the relevant papers/documentation if their contents largely resembled some other publications of the same author(s) that we have already analyzed. The relevant numbers are shown in Table 1. Note that they may increase in the future, as new support may be collected for the patterns discovered. The table shows the numbers only for the patterns that we have discovered so far. Other patterns exist as well, and their discovery is the subject of our on-going work.

Table 1
Statistics about the ITS-architectural patterns discovered so far

Total number of patterns discovered, at all levels of abstraction	12
Total number of papers analyzed	96
Total number of other relevant sources analyzed (e.g., system documentation, interviews with the system architects, implementation details)	13
Total number of ITSs found to support some of the patterns that have been discovered (the system status -prototype, tested, used in practice, etc. -was excluded from the analysis)	63
Number of ITSs that use individual patterns (again, the system status -prototype, tested, used in practice, etc. -was excluded from the analysis)	5-12

Knowledge of more general software patterns has been very useful in the process of discovering ITS-related patterns. Some patterns that we have discovered can be seen as close relatives of some more general design patterns, patterns for software architectures, or analysis patterns. This is inevitable -software patterns are plentiful, and it is hard to discover something completely new in the architectures of systems developed in some specific domain, such as ITS/AIED. However, we have tried as much as possible to concentrate on ITS-specific architectural issues and stay as much away as possible from general software patterns. Hence all the patterns we have discovered are at least to an extent specific to ITS architectures, and for each one of them we have also recorded what general software patterns are related to it.

CLASSIC ITS ARCHITECTURE

The first pattern we describe here should look familiar to most of the readers. However, it is presented here using the template described in the Introduction. It should allow the reader to get

used to the template, since all the other patterns presented in the subsequent sections use the same template for consistency.

Name. Classic ITS architecture.
Category. Analysis pattern.
Context. Specification of typical competences that an ITS provides.
Problem. Define ITS modules and their functionalities according to the competences the ITS provides.
Forces. The modules should reflect the ITS domain and pedagogical expertise, as well as its capabilities of driving the learning session and interacting with the student in a personalized way.
Solution. In the classic ITS architecture, Figure 1 (Wenger, 1987), functionalities of its modules are defined according to the competences the ITS provides:

- the *Expert Module* represents the domain competence of the ITS, that is the ability to solve problems within the domain;
- the *Student Module* contains the diagnostic competence of the ITS and generates the student model with all information about the individual learner;
- the *Tutor Module* is responsible for the instructional competence; this module provides implementations of different tutoring strategies;
- the *Communication Module* implements the human-computer-interface of the ITS. To enable the user to interact with the ITS this component has to represent some knowledge about how to interact, that is communication competence.

These modules interact with each other in many ways: the Tutor Module chooses a proper learning task based on information from the Student Module; the Communication Module presents the task and converts user input to a format that can be processed further by the Expert Module, which tries to make a diagnosis of the student´s problem solving behaviour. This diagnosis is used for an update of the Student Module and so on.

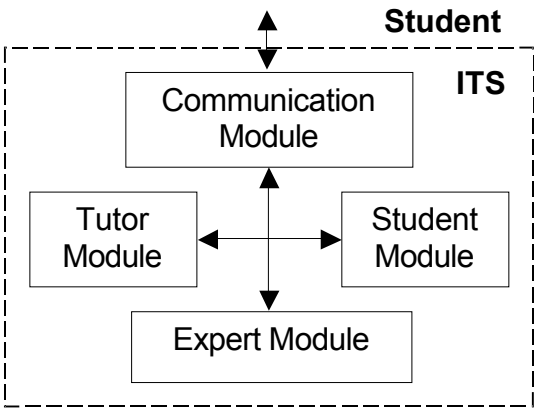

Fig. 1. Classic ITS architecture (system border in dotted lines).

Diagram. See Figure 1. This competence-oriented decomposition of ITSs is presented in detail in (Wenger, 1987).

Variants. Even though there were other reference architectures presented in recent years (which we present later), this *Classic ITS Architecture pattern* is still very popular and can be found both in many traditional ITSs and in various recent ones, such as SYPROS (Harrer & Herzog, 1999), SQL tutor (Mitrovic, 2003) and SlideTutor (Crowley et al., 2003). Its importance is also recognized by the IEEE Learning Technology Standards Committee, which proposes a very similar structure for learning systems in its architecture specification draft (Farance & Tonkel, 2001). Their four main components, called processes, have functionalities corresponding to those of the ITS-modules -the components are *Evaluation* (Expert Module), *Learner Entity* (Student Module), *Coach* (Tutor Module), and *Delivery* (Communication Module).

In order to illustrate how this pattern has evolved in numerous ways over the years and how it is still replicated in recent projects, Figure 2 shows an example. It represents the architecture of the Code Tutor ITS, which is developed to help the students learn the basics of radio-communications and is actively used in practice since 2002 (Shimic & Devedzic, 2003). Note that in addition to the "instantiations" of traditional modules -*Student model*, *Tutor*, *Expert module*, *GUI* -this architecture also includes several modules necessary to enable the use of Code Tutor in Web classrooms -*Web server*, *Servlet engine*, *Session monitor*. Moreover, the figure also indicates the use of different GUIs for the teacher and the students, as well as an embedded expert system component in the Expert module.

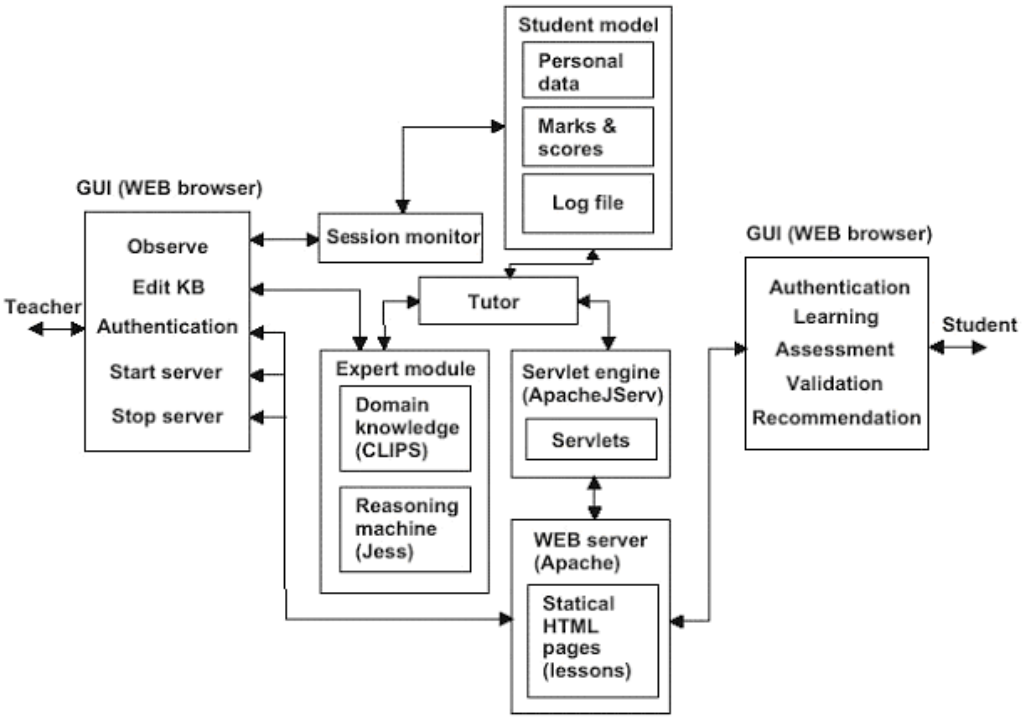

Fig. 2. Architecture of the Code Tutor system (components and collaborations).

Related patterns. Patterns for distributed ITS architectures and for Web-based ITS architectures.

KNOWLEDGEMODEL-VIEW ARCHITECTURE

Name. KnowledgeModel-View.
Category. Design pattern.
Context. Reference architecture for ITSs.
Problem. Find a general architecture for ITSs that takes into account different variants of the classic ITS architecture.
Forces. In recent years there were several proposals for reference architecture of ITS other than that described by the *Classic ITS architecture* pattern.
Solution. In our efforts of "*pattern mining*" or pattern discovery, we found an architectural pattern for ITSs that generalizes two proposals for reference architecture for ITSs, the "integration-oriented" architecture from Peter Brusilovsky (Brusilovsky, 1995) and an alternative proposal from Meike Gonschorek (Gonschorek, 1998). An exhaustive analysis has shown that a large number of current ITSs (such as in (Brusilovsky 2003)) can be seen as instances of this pattern. *KnowledgeModel-View* means that the architecture has distinct components for a) all kinds of *knowledge models* within an ITS, and b) components for presentation of information for the user and interaction between user and system (these can be called *views*). Views are updated when the content of a knowledge model changes, so we have a typical interaction protocol between models and views, which is called "Publish-Subscribe" (Gamma et al., 1995). So the basic structure of this ITS-pattern is very similar to the general software engineering pattern "Document-View" (Buschmann et al., 1996), consisting of one central document that contains all the relevant knowledge/data, and several views, which are updated every time the document changes. The difference is that in the ITS-context we may have multiple sources of knowledge (student model, expert model, tutoring rules) and therefore not a *1:n* relation but a *m:n* relation between models and views.
Diagram. The typical structure diagram for this pattern is shown in Figure 3.

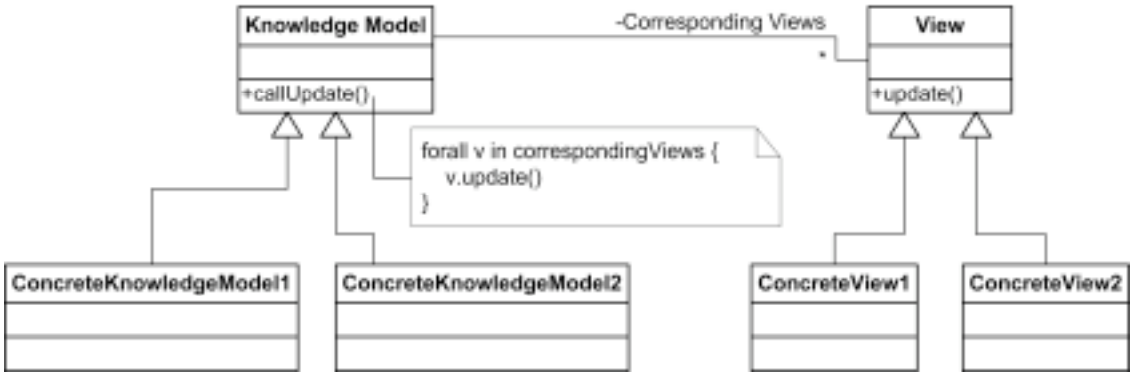

Fig. 3. UML class diagram of KnowledgeModel-View pattern.
(* indicates any number of class instances)

Variants. As we already mentioned, this pattern generalizes two architectures proposed in the literature. In (Brusilovsky, 1995) the "student-model-centered" architecture consists of a central student model (that is one knowledge model of our pattern), agents and their knowledge, and so

called "tools with interfaces" that enable the interaction with the learner (these are the views). Both agents and tools use a projection of the student data that is created by "projectors". The projectors can either be considered part of the central student model (our pure pattern) or as components that provide a Pipes-and-Filters pattern (Buschmann et al., 1996) between model and views (in contrast to the usual pipes and filters which is unidirectional this one is bi-directional, i.e. data flow from model to tool or the other way round). Figure 4 sketches this architecture:

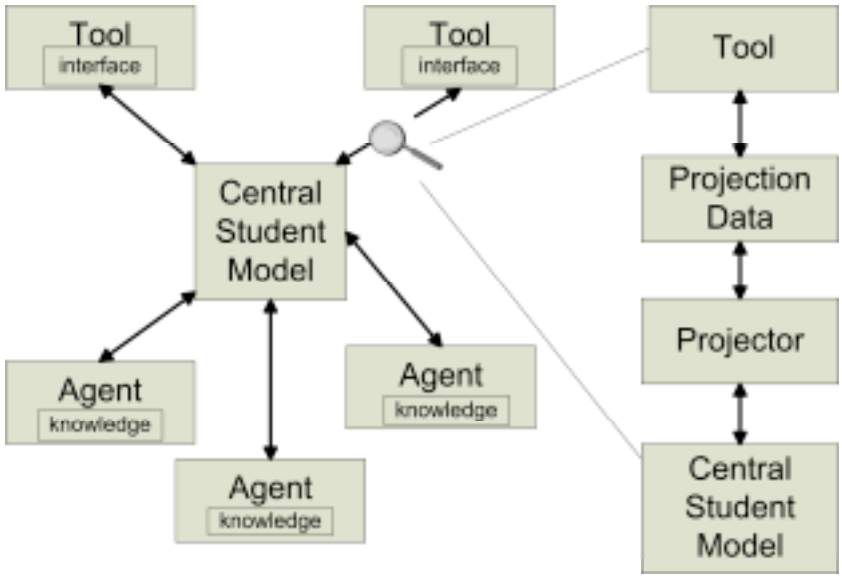

Fig. 4. Student-model-centered architecture (according to (Brusilovsky, 1995)).

The "integration-oriented" architecture in (Gonschorek, 1998) stresses the decomposition of components in ITSs into a knowledge module (which consists of several different knowledge models), presentation components (views for the representation of information for the user, which just present but do not process user input), and a control module that defines the reactive part of an ITS, uses information from the knowledge module, initiates system actions and changes to the knowledge bases. In interactive systems the control module is called controller, and the whole architecture Model-View-Controller architecture (Buschmann et al., 1996). So the integration-oriented architecture is an architecture in a model-view-controller-like style that is specifically tailored to ITSs with several knowledge models. The detailed architecture can be seen in Figure 5.

Related patterns. We call this general pattern for ITS architectures "KnowledgeModel-View" pattern as a reference to the general architectural patterns "Model-View-Controller" and "Document-View" (Buschmann et al., 1996), because there are some similarities between these general architectural patterns and the KnowledgeModel-View pattern. The Model-View-Controller pattern in relation to collaborative learning systems is also discussed in (Suthers, 2001).

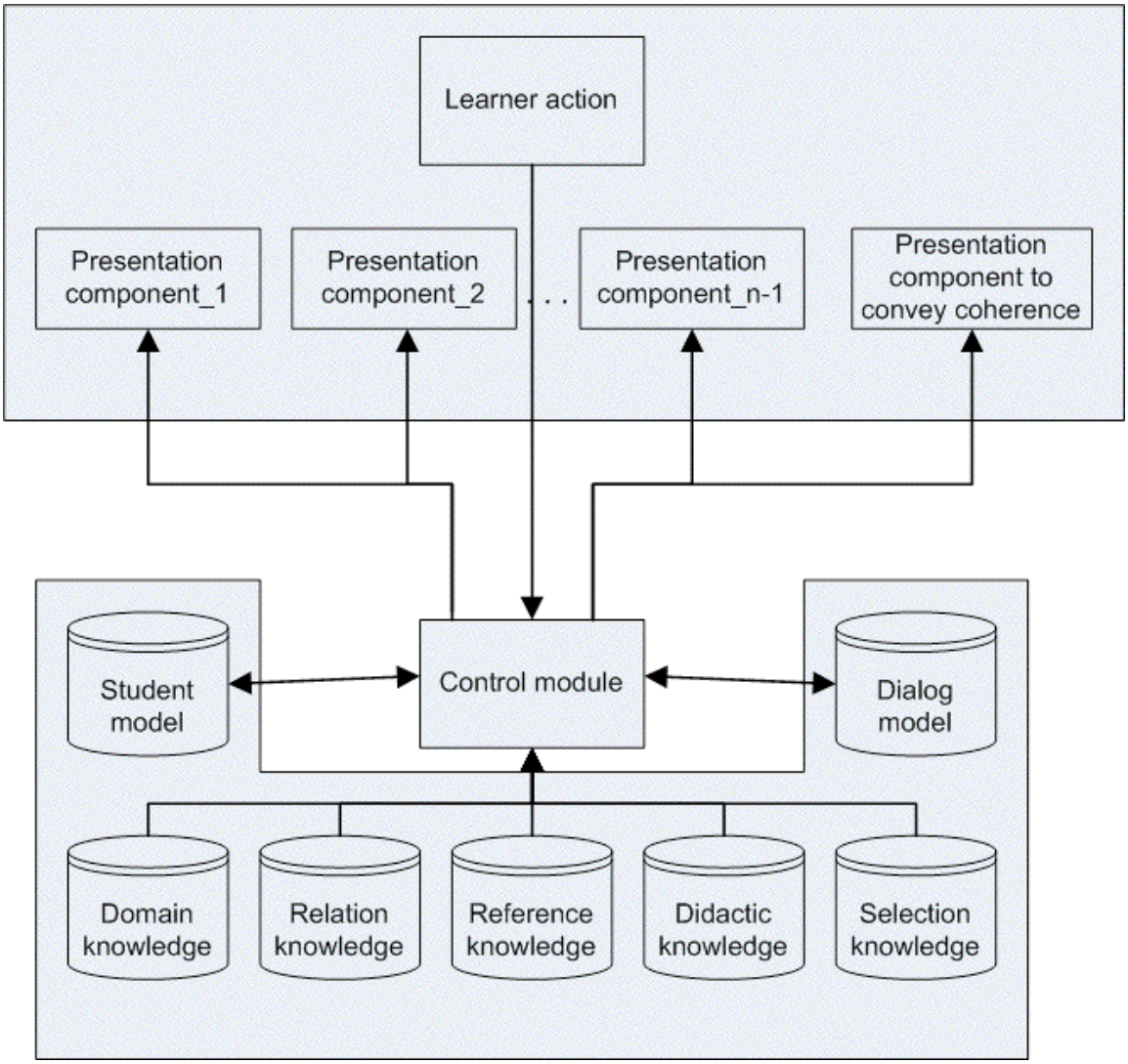

Fig. 5. Integration-oriented architecture (according to (Gonschorek, 1998)).

PATTERNS FOR COLLABORATIVE LEARNING SYSTEMS

Name. ICSCL student model.
Category. Design pattern.
Context. Student modelling in Computer Supported Collaborative Learning (CSCL) systems.
Problem. Design a suitable model to enable representing both an individual learner as a participant to a group of learners in collaborative learning situations, and the group of learners as a whole.

Forces. CSCL systems enable different distributed learners to learn together. CSCL systems that try to support the resulting learning communities intelligently (ICSCL) with techniques used in ITSs for single learners, such as adapting the learning material to the needs of the groups or specific members, have to model the group as well as the individual learner.

Solution. The raw material for student and group modelling can be found on different levels:

- *domain level*, where all the problem solving activities take place;
- *conversational level*, where the cooperation, division of labour, and discussion of the learning community takes place.

For each level an ITS may provide several editors accepting user input and creating raw data. The generated raw data is task-focused action at the domain level, while the raw data at the conversational level is coordination-oriented interaction information (mechanisms to trace the interaction in ICSCL are often based on dialogue games or patterns of social interaction represented as conversational networks (Harrer, 2001)).

The raw data (domain level actions, speech acts…) can be transformed into more abstract learner model information (such as mastery of a specific learning goal or participation of one learner within a community), which again can be separated into two distinct types of models:

- *individual learner models* that contain all the data relevant to one specific learner (knowledge state, motivational traits, learner type);
- *group models* that contain all the data relevant for the learning community (complementary and conflicting knowledge, specific roles within the community, relations between the members).

The transformation of raw data into the learner model contents can be done "on the fly" (that is just when the raw data is generated) or at discrete points of time (such as the end of a group learning session).

Diagram. If we want to consider all of these aspects in the structure of the student modelling subsystem and to provide a flexible and easily extendable interface, we get a design like the one in Figure 6.

Variants. *ICSCL student model* pattern is inspired by the ideas of Ana Paiva (Paiva, 1997) and systems akin to Intelligent Distributed Learning Environments (IDLE) presented in (Harrer, 2000). Recently the interrelations of individual and group model information have been discussed in (Winter & McCalla, 2003).

Related patterns. The design presented in Figure 6 contains some class hierarchies which are structurally similar. For each level of user action (domain level and conversational level) we have editors and raw data, for each type of learner model (individual and group model) there are the models and the contained learner model contents. The hierarchies are connected via the class hierarchy of raw data processors. This design resembles the *Abstract Factory* design pattern (between the upper two hierarchies), as well as the *Factory Method* design pattern (processor and content hierarchies) (Gamma et al., 1995). The structure can be easily extended when additional editors or processing components are brought into the system.

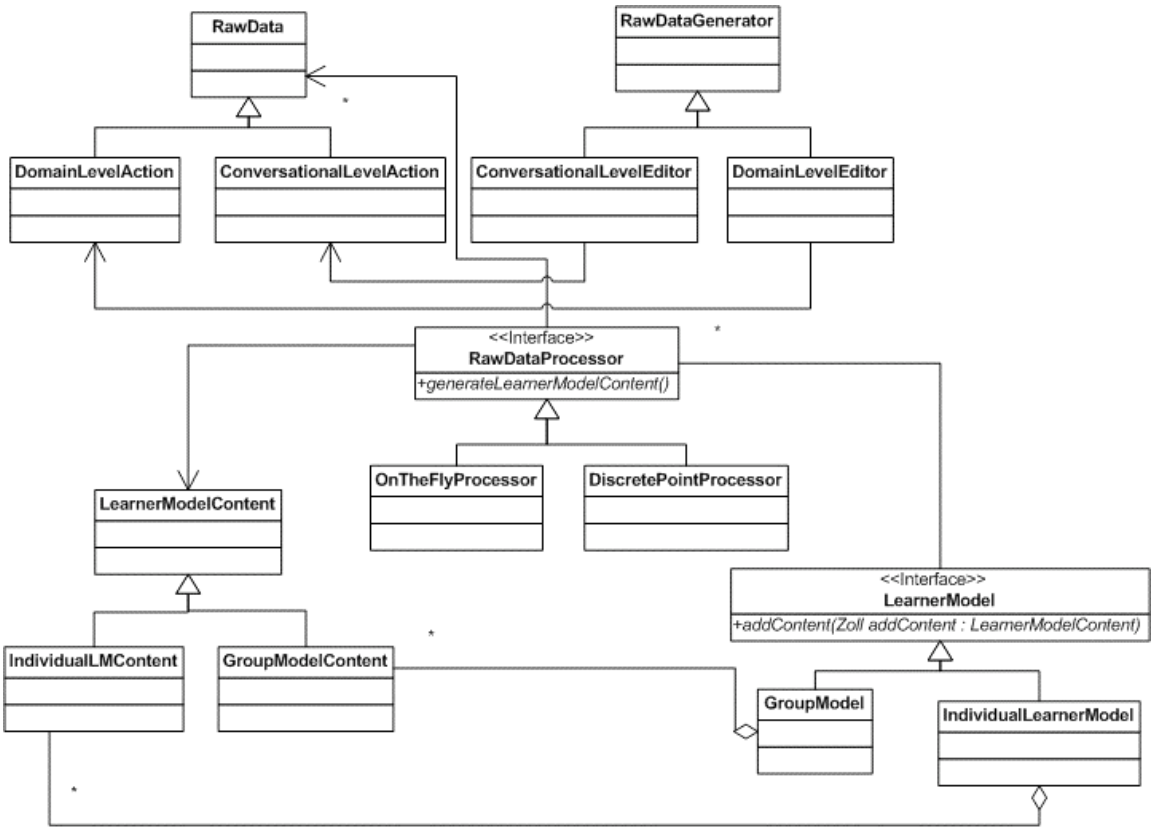

Fig. 6. UML class diagram of ICSCL student modeling pattern.
(* indicates any number of class instances)

PEDAGOGICAL AGENTS AND SOFTWARE PATTERNS

Over the last several years there has been significant interest in the ITS research community for applying intelligent agents in design and deployment of ITSs. The focus has been on *pedagogical agents*, i.e. autonomous agents that support human learning by interacting with students in the context of interactive learning environments (Johnson et al., 2000). Pedagogical agents monitor the dynamic state of the learning environment, watching for learning opportunities as they arise. They can support both collaborative and individualized learning, because multiple students and agents can interact in a shared environment. Pedagogical agents can provide numerous instructional interactions with students, promote student motivation and engagement, and effectively support students' cognitive processes. Some of them are represented by animated characters (Rickel, 1999) that give the learner an impression of being lifelike and believable, producing behaviour that seems natural and appropriate for the role of a virtual instructor or guide. In distributed learning environments, multiple pedagogical agents can collaborate in different ways to help students in various learning activities.

We have analyzed a number of pedagogical agents described in the ITS literature, as well as some of them whose demos are available on the Web. The idea has been to see to what extent people designing pedagogical agents and multiagent systems use patterns (implicitly) in their design. The two patterns shown here are related to the design of individual agents only. Note, however, that patterns exist also in different designs of multiagent educational systems.

Name. Generalized Pedagogical Agent (GPA).
Category. Analysis pattern.
Context. Internal structure of pedagogical agents, such as modules and data/information flows. Does the structure reflect the role of the pedagogical agent and how?
Problem. Given the fact that pedagogical agents in ITSs can play different roles (learners, teachers, companions, assistants, and so on) in different educational settings, abstract the general structure and show how such a generalization supports the different roles.
Forces. Recognizing the essential functions of specific modules in the architecture of a certain pedagogical agent can be difficult because of a number of applications of pedagogical agents and abundant design specifics.
Solution. We have found that numerous designers of pedagogical agents essentially follow the pattern shown in Figure 7, which we call *General Pedagogical Agent pattern*, or *GPA pattern*. The name comes from the fact that the pattern has been abstracted out of a number of agents playing different roles in different educational settings (see above), yet having much in common. Boxes and data/knowledge flows represented by solid lines have been identified (under various names) in all the agents we have analyzed. Those represented by dashed lines have been found in a number of agents, but not in all of them.

The first mandatory participant in the GPA pattern is *Knowledge Base* that can include domain knowledge, pedagogical (instructional, tutoring) knowledge/strategies, and student model. Some of the examples include the Learning Tutor agent (Hamburger & Tecuci, 1998), Disciple agents (Tecuci & Keeling, 1999), pedagogical actors (Frasson et al., 1997; Frasson et al., 1996), animated character agents (Marsella et al., 2003) and so on. Only in the design of Disciple agents the *Knowledge Base Manager* is shown explicitly, although textual descriptions of most of other pedagogical agents also indicate its presence. *Knowledge Base Manager* helps the other participants use and update all kinds of knowledge stored in *Knowledge Base*.

Each pedagogical agent also has some kind of *Problem Solver* (i.e. *Logic*, or *Reasoning Engine*) that can include inferencing, case-based reasoning, and other tasks normally associated with learning and teaching activities. For example, Learning Tutor agent has both a tutoring engine and an inference engine (Hamburger & Tecuci, 1998). The problem solver of pedagogical actors is essentially a control module and a decision maker that decides on whether the actor should take an action, how to do it, on what stimulus it should react, and the like (Frasson et al., 1997). Teachable Agent developed by Brophy et al. has explanation and simulation generators in the problem solving component (Brophy et al., 1999).

The *Communication* (*Interface*) participant is responsible for perceiving the dynamic learning environment and acting upon it. Typically, perceptions include recognizing situations in which the pedagogical agent can intervene, such as specific student's actions, co-learner's progress, and availability of desired information. Examples of typical actions are "display a message", "show a hint", and "update progress indicator".

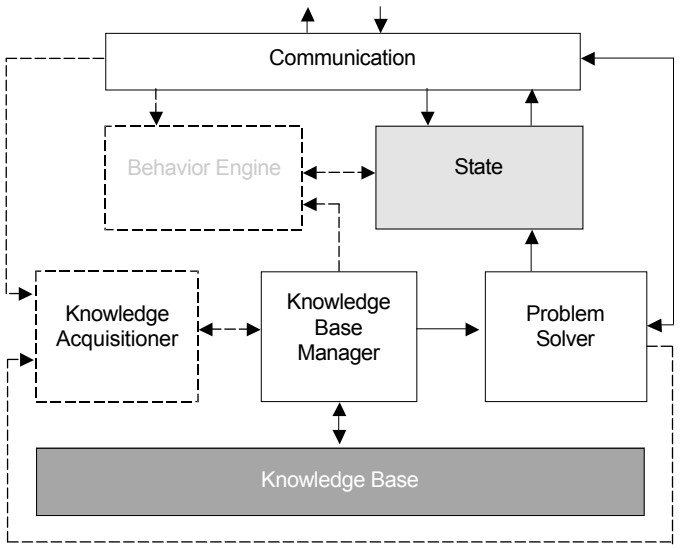

Fig. 7. The GPA pattern (dashed lines show optional elements, shaded boxes components with highly variant specifications).

State is a general abstraction for many different kinds of states a pedagogical agent can be made "aware of", as well as both volatile and persistent factors that influence the states. It can refer to the agent's expression, expressiveness, emotions, and personality, as in the Classroom Agent Model (Yin et al., 1998). It can also represent the agent's different mental states and verbal behaviour, as in the case of Vincent, the pedagogical agent for on-the-job training (Paiva & Machado, 1998). Furthermore, *State* can contain current values of parameters of the agent's relationships with other agents -the relationships that the agent is able to create, reason about, and destroy, all according to its goals. This is the case of Social Autonomous Agents (Vassileva, 1998). Teachable Agent's state includes its disposition, or learning attitude, which may for example determine whether the agent will learn by picking a quick, reasonable solution, or will rather spend more time in order to come to a precise solution (Brophy et al., 1999).

Occasional participants in the GPA pattern include *Knowledge Acquisitioner* and *Behavior Engine*. *Knowledge Acquisitioner* reflects the fact that apart from reactive, reasoning, and decision-making capabilities, cognitive pedagogical agents also possess a learning capability that makes possible for the agent to update and modify its knowledge over time, using some machine learning technique(s). For example, the Learning Tutor and Disciple agents include a machine learning and knowledge acquisition component explicitly (Hamburger & Tecuci, 1998), (Tecuci & Keeling, 1999). The cognitive layer of pedagogical actors provides them with self-improvement capabilities based on the analysis of the student's performance in different learning situations and machine learning techniques (Frasson et al., 1997; Frasson et al., 1996). Essentially, it categorizes different learners using conceptual clustering techniques and builds a case base of situations. Self-improvement is done by using case-based reasoning to adapt the situation to a new case. In a similar way, cognitive agents can provide adaptivity and flexibility in learning in terms of autonomously establishing learning objectives, and creating, locating,

tracking, and reviewing learning materials, such as diagnostic instruments, scenarios, learning modules, assessment instruments, mastery tests, etc. (Canut et al., 1999).

If present in the design of a pedagogical agent, *Behavior Engine* (*Expression Engine*) is responsible for analysis of the agent's current internal state and possible modifications of (parts of) that state. In general, *Behavior Engine* depends on the current perception of the agent's environment supplied by the *Communication* module, the current *State*, and possibly on some heuristics specified in the *Knowledge Base*. Its output is changes in the current *State* that are then converted into appropriate actions by the action part of the agent's *Communication* module. Frequently used instances of *Behavior Engine* include Emotion generator and Behavior generator such as those included in the Classroom Agent Model (Yin et al., 1998), Emotive Behavior Sequencing Engine of lifelike pedagogical agents (Lester et al., 1999), physical behavior generator (i.e., layout handler, or the "Body") and dialog handler such as those of Vincent (Paiva & Machado, 1998), and social behavior generator in pedagogical agents participating in multiagent systems (Vassileva, 1998). This last instance of *Behavior Engine* is always used for pedagogical agents participating in collaborative learning and collective decision-making processes.

Diagram. See Figure 7.

Variants. As an illustration of how GPA is used in practical agent design, consider the Classroom Agent Model (Yin et al., 1998), shown in Figure 8. Classroom Agents represent students in a classroom, each one having its own personalities and emotions (*Personality* and *Emotion* are, in fact, instances of *State*). *Learning* is the word that the Classroom Agents Model authors use to denote all kinds of problem solving activities associated with learning situations (an instance of *Problem Solver*). In these agents, the *Behavior Engine* is split into two distinct parts, one to generate the agent's emotions (*Emotion Generator*), and another one (*Behavior Generator*) to formulate the agent's learning actions and pass them to the effector (the *Action* part of the *Communication* module). The idea is that for a learning agent perceiving an event that motivates it (e.g., an easy exercise) positive emotions will be generated and will promote its learning (and vice versa), which will certainly be reflected in *Behavior Generator* when formulating the agent's next actions. *Behavior Engine* is optional in the GPA pattern (dashed line for the corresponding box in Figure 7), but certainly does appear in the Classroom Agent Model (solid lines for *Emotion Generator* and *Behavior Generator* in Figure 8).

We have also identified several other variants of the GPA pattern. For example, the pedagogical agent of the Adele system downloads its case task plan and an initial state, as well as a student model, from the server, and only the reasoning engine and animated persona are on the client side (Shaw et al., 1999). Another variant can be seen in Vincent, where problem solving is done in the agent's Mind component (Paiva & Machado, 1998). Vincent's Mind is a combination of a knowledge handler and a dialog handler -a kind of a mixture of functionalities of *Problem Solver* and *Behavior Engine.*

Related patterns. Further abstraction of GPA-based agents can lead to the conclusion that they are essentially instances of *reflex agents with states*, *goal-based agents*, and *knowledge-based agents*, as higher-level abstractions of all intelligent agents (Russell & Norvig, 1995). Such a conclusion would be true -*State* corresponds to the agent's internal state in all these higher-level abstractions, *Problem Solver*, *Knowledge Acquisitioner*, and *Behavior Engine* reflect the agent's general ability to make inferences, decisions, and updates of its knowledge, and so on. However, going that far would take us completely out of the scope of ITSs to much more general AI

problems. That would, in turn, mean loosing the sense of context where the pattern applies - remember, a pattern is a generalized solution of a typical problem within a typical context.

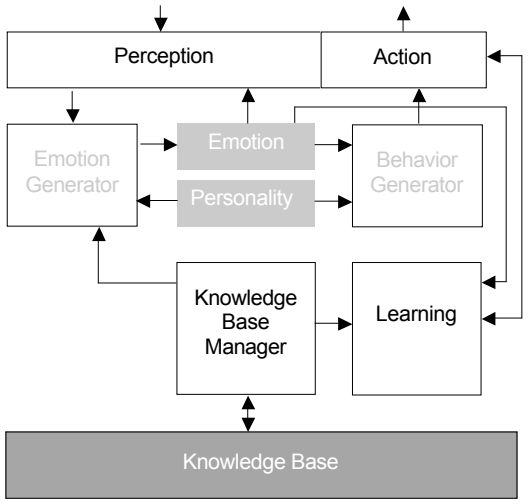

Fig. 8. An example of using the GPA pattern in the Classroom Agent Model (after (Yin et al., 1998)).

CO-LEARNER PATTERN

Name. Co-Learner.
Category. Analysis pattern.
Context. Introducing an artificial learner (a program) acting as a peer in an ITS has proven to have a number of positive effects on the learner. It ensures the availability of a collaborator and encourages the student to learn collaboratively, to reflect on and articulate his past actions, and to discuss his future intentions and their consequences (Goodman et al., 1998). Artificial learners can take different roles, such as:

- *learning companion* (Chan & Baskin, 1988), which learns to perform the same learning task as the student, at about the same level, and can exchange ideas with the student when presented with the same learning material;
- *troublemaker* (Aimeur & Frasson, 1996), which tries to disturb the student by proposing solutions that are at times correct, but are wrong at other times, thus challenging the student's self-confidence in learning;
- several *reciprocal tutoring* roles, as described in (Chan & Chou, 1997).
- roles that aim to stimulate collaboration and discussion within learning communities, such as *observer* (Dillenbourg et al., 1997), *diagnostician* or *mediator* (Harrer, 2000).

Problem. Characterize the functions, interactions, and dataflows typical for artificial learners, regardless of the roles they may take.

Forces. In all the cases described in the Context section we can talk about a distinct learning paradigm that often goes under different names -*learning companion systems*, *co-learner systems*, *simulated students*, and so on. We prefer the term co-learner. It is important to note that *architecturally*, all ITSs involving a co-learner have much in common regardless of the role the co-learner takes.

Solution. The *Co-Learner pattern*, reflecting the learning paradigm just mentioned, is shown in Figure 9. The classic 3-agents triad -*Tutor-Student-Co-Learner* -is shown in Figure 9a with more details than the literature on co-learners usually offers, due to the fact that all pattern diagrams have to show both the participants and their communication paths clearly. Hence that part of the figure stresses the "who communicates with whom" and "what knowledge and data are involved" issues. Moreover, since this is an architectural view, it is necessary to show details more-or-less irrelevant for instructional aspects of co-learner systems. For example, the *System* component acts as a supervisor and performs all the control and scheduling of activities of the three major agents. It is shown in grey, though, since it doesn't contribute essentially to the major knowledge and information flows. Furthermore, Figure 9a clearly indicates what kinds of knowledge and data each agent needs. Put this way, it turns out that the Co-Learner pattern belongs, in fact, to *blackboard architectures* (Buschmann et al., 1996) -Wang and Chan note that explicitly (Wang & Chan, 2000). All knowledge and data are on the blackboard, but usually only the *Tutor* agent accesses all of them. *Student* and *Co-Learner* normally have access only to the *Learning Task* part of the blackboard (thick data-flow lines). Variants are discussed in a later paragraph (dashed data-flow lines).

If *T*, *S*, and *C* denote the *Tutor*, *Student*, and *Co-Learner* agents, then their communication in the Co-Learner pattern, Figure 9a, is as follows:

- T → S -present learning task and materials, explain the format of learning activities, generate problems, provide guidance, advice and hints, generate examples, evaluate solutions, generate final justifications of the solution and/or insightful retrospective comments, negotiate with the student (Chan & Baskin, 1988). Most of this communication goes through the *Learning Task* part of the blackboard, while all the necessary domain knowledge is in the *Domain Knowledge* part. *Tutor* can use different *Teaching Strategies*, and during the course of the learning task it develops the *Student Model*.
- S → T -ask for clarifications & help, present solutions, request directions and references for future learning. Most of this communication goes through the *Learning Task* part of the blackboard.
- T → C -much like T → S for the learning companion role of the *Co-Learner* agent, but can vary a lot for other roles. For example, if *Co-Learner* is a troublemaker, it doesn't get much instruction or direction from the *Tutor*. On the contrary, troublemaker usually has access to *Domain Knowledge* as much as *Tutor*. Troublemaker has the level of competence superior to that of the student, in order to engage him. It is only disguised as a co-learner, while its role is pedagogically different. However, if *Co-Learner* is a true learning companion, it performs the learning task in much the same way as the student (Chan & Baskin, 1988), and during the course of the learning task *Tutor* develops the *Co-Learner Model*.

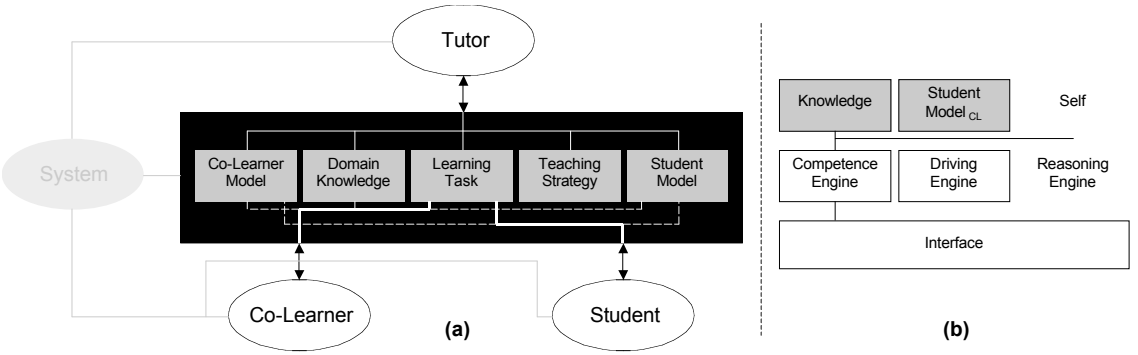

Fig. 9. Co-Learner pattern a) communication paths b) inside the Co-Learner.

- C → T -usually a highly restricted version of S → T, since the effects of co-learner's learning are of much less importance than those of student's learning. However, in some variants of reciprocal tutoring this communication can be much more important and much more elaborated (Chan & Chou, 1997).
- S → C -observe (watch) the co-learner working on the learning task individually, ask for assistance, give suggestions when asked, decide on problem-solving strategy, explain strategy, clarify problem, compare solutions, discover mistakes, correct mistakes (Brophy et al., 1999), (Chan & Baskin, 1988), (Goodman et al., 1998). Also, in different variants of reciprocal tutoring, much of T → S and S → T (Chan & Chou, 1997). An important consequence for system designers follows from that fact -all three major agents can be derived from the same, more general pedagogical agent, and should have similar functions (minor differences can be easily implemented using polymorphism).
- C → S -much like S → C for learning companions, but also much like T → C for other roles. For an excellent survey of co-learners' roles and activities in communication with human learners, see (Goodman et al., 1998).

Figure 9b shows a fairly generalized version of the Co-Learner agent itself. Functionally, it is much like GCM (General Companion Modeling) architecture of Chou et al. (Chou et al., 1999). However, it stresses important details of Co-Learner's internal structure in terms of an analysis pattern. Note that, in general, Co-Learner can be a "mini-ITS". Depending on the role, it can have more or less of its own *Knowledge*, both domain and pedagogical. If Co-Learner has little knowledge, it is a novice learner; if its knowledge is comparable to *Tutor*'s, it is an expert. That knowledge can grow over time for teachable agents (Brophy et al., 1999), and in all reciprocal-tutoring cases when Co-Learner switches the teaching and learning roles with another agent in the system. For troublemakers, that knowledge can include details of the learning by disturbing strategy (although it is possible, in principle, to access it on the blackboard as well). Co-Learner constructs and maintains its own model of the human learner's knowledge and progress, *Student Model* $_{CL}$, which is generally different from the *Student Model* built by *Tutor.* Also, the agent's own internal state (attribute values, learning status, the level of independence,

motivation, personality characteristics, the corresponding animated character (if any)) is stored in *Self*. Multiple co-learner systems can have Co-Learner agents with different *Self* characteristics.
Diagram. See Figure 9.
Variants. In practice, the Co-Learner pattern has many variants. Note, for example, that *Student* and *Co-Learner* can optionally access some parts of the blackboard other than *Learning Task* (the dashed lines in Figure 9a). Troublemakers have access to *Domain Knowledge* (Aimeur & Frasson, 1996; Frasson et al., 1997), and some authors have studied the benefits of letting the student access the *Student model* and the *Co-Learner Model* as well (e.g., (Beck et al., 1997; Bull et al., 1999)), an approach called inspectable student models. Also, if Co-Learner is a troublemaker then the student explains his decisions to the troublemaker in a process controlled by the *Tutor* agent directly, and not by the *System* (Aimeur et al., 1997). The *Tutor* can even be omitted from the system, as in teachable agents (Brophy et al. 1999), in the similar "learning by teaching the learning companion" strategy (Scott & Reif, 1999), and in some other kinds of reciprocal tutoring (Chan & Chou, 1997). Even group learning can be represented by the Co-Learner pattern (at least to an extent), by letting the system have multiple learning companions with different competences and different personas (peer group learning) (Hietala & Niemirepo, 1998). Alternatively, the system can have multiple *Tutors* with different personas and the student can learn from them selectively, and *Co-Learner* can be the personal agent of another human learner on the network, as in distributed learning (Chan & Baskin, 1988). The Social Intelligence Project (Johnson et al., 2003) aims at the development of pedagogical agents, that take into account the affective and motivational state of the learner for the human-agent interaction.
Related patterns. Figure 9b is also an instance of the GPA pattern, Figure 7. Co-Learner's *Knowledge* corresponds to *Knowledge Base* in GPA, while *Student Model* $_{CL}$ and *Self* are instances of GPA's *State*. Likewise, Co-Learner has a *Reasoning Engine* (an instance of *Problem Solver*), capable of simulating various activities in learning and teaching tasks, depending on the role Co-Learner plays (see C □ S and C □ T communication above). In fact, *Reasoning Engine* fully corresponds to the *Learning task simulation module* of GCM architecture, described in (Chou et al., 1999). *Driving Engine* (*Behavior Engine* of GPA pattern and *Behavior module* of GCM architecture) is responsible of generating specific behaviour that drives the agent in playing a specific co-learning role (a learning companion, a peer tutor, and the like). *Competence Engine* (*Knowledge Acquisitioner* in GPA) is Co-Learner's mechanism to increase its competence by acquiring new knowledge, i.e. to learn and modify contents of its knowledge bases accordingly. For example, learning companions can employ machine-learning techniques or simulated knowledge acquisition (controlled externally, by the *System*, in order to adapt the companion's competence to the student's current knowledge level) (Chan & Baskin, 1998).

PATTERN LANGUAGES AND ITS

Patterns never exist in isolation -all knowledge and application domains are usually characterized by a number of patterns. When several related patterns in a given domain are discovered, named, described, and woven together in a collection, they form a pattern language for that domain (Schmidt et al., 1996; Schmidt et al., 2000). A pattern language is *a network of patterns* that call upon one another. Patterns in a pattern language can be used *in combination* to create solutions. The coverage of a domain by a corresponding pattern language can vary; yet each pattern

language reflects a number of important issues in the domain (domain knowledge), thus providing specialists with vocabularies for talking about specific problems.

So far, there has been only one attempt to formulate a pattern language in the domain of ITS. A small collection of related ITS-architectural patterns has been discovered and described in *PLAIT*, a Pattern Language for Architectures of Intelligent Tutors (Devedzic, 2001). That initial collection of patterns represents only the core of PLAIT focused on layered ITS architectures. The language evolves and continues to accumulate new patterns, as they get discovered over time.

For illustration purposes only, Figure 10 shows topologies of some of the patterns defined in PLAIT. In describing an architectural pattern topology is not all there is, but is certainly a major issue in giving an initial idea of what the pattern is about and understanding how it is used in designing the architecture of practical systems in the domain. Shaded boxes in Figure 10 represent the architectural modules (components) belonging to the patterns themselves, whereas the other boxes are there to indicate how the patterns fit into the overall architecture (but are not parts of the patterns). The arrows represent the data flow paths between the components. See (Devedzic, 2001) for complete and detailed descriptions of the patterns from Figure 10.

The patterns in PLAIT condense parts of ITS-architectural design knowledge of experienced designers and make it explicit and available to others. In fact, each pattern in PLAIT indicates a good practice in designing architectures of ITSs, i.e. a practice that has proved sound and efficient. By reusing these patterns, developers of future ITSs can benefit from previous experience and create solutions to new problems without "reinventing the wheel". Therefore, the patterns in PLAIT are a medium that helps the dissemination of best practices within the domain of ITS architectures.

The lessons learned through development of PLAIT let us better understand what it takes to develop a full-fledged architectural pattern language for ITSs. A major constraint introduced on purpose in the initial version on PLAIT -the focus on *layered* architectures only -has made PLAIT's initial domain too narrow. Although the patterns described in the initial version are all useful in their respective design contexts, many other types of ITS architectures were not covered at all. Also, they were all essentially *design* patterns, concentrating mainly on lower-level issues in ITS-architectural design. Hence the patterns in PLAIT's initial collection did not contain a number of ITS-specific issues. The conclusion we drew was pretty straightforward. First, in spite of the fact that a narrow domain makes it easier to discover patterns, the domain to focus on in developing a pattern language should still be large enough as to allow for a wider ITS-architectural context in which the patterns from the language apply. Moreover, the context should be more ITS-specific, such as Web-based ITS architectures, or CSCL architectures, to name but a few. Second, in order to ensure that the language will be more widely accepted, higher-level architectural issues should also be covered. Hence searching for *analysis* patterns in ITS architectures and incorporating them in the language is a must.

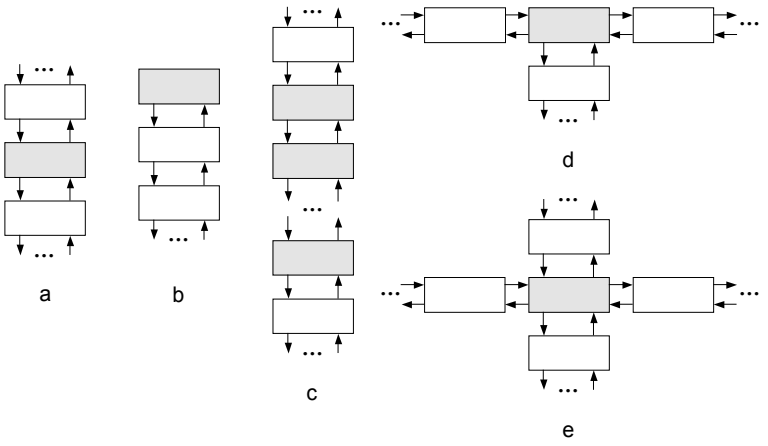

Fig. 10. Some patterns from PLAIT (each with the characteristic component(s) shaded)
a) the *Inserted Layer* pattern b) the *Top* pattern c) the *Cascade* pattern
d) the *T-join* pattern e) the *Cross* pattern.

Important implications of these observations for ITS research in general are as follows. Good candidate domains for formulating pattern languages are subfields of ITS (CSCL, pedagogical agents, learner modelling, etc.). Subfield specialists should take care of identifying patterns in the subfields, naming the patterns, and describing how they relate to each other. Pattern languages are often visualized in diagrams showing the relations among the patterns graphically (Coplien & Zhao, 2005). Pattern names in a pattern language should be descriptive enough to the researchers and developers in the subfield. In other words, there should be no need for the researchers to read detailed descriptions of the patterns in the language in order to have an intuitive understanding of what issues the patterns actually cover. After reading a general description of the pattern language, seeing the diagram that visualizes the language, and knowing the names of the patterns in the language, researchers in the subfield should be able to tell what a specific pattern relates to and how to apply it. However, pattern topologies and names are important only as guidelines and orientation; in practical developments, details are necessary in order to analyze the design, to apply the patterns, and to consider alternatives. With some experience over time, the names of the patterns get accepted by the specialists and become parts of the subfield's overall vocabulary.

DISCUSSION

ITSs are, after all, just another kind of *software* system, hence their design should conform to well-established general rules and practices of software design. This is not to say that ITS-specific features should be neglected in their design -we advocate that it is necessary to integrate them with general software design issues.

Still, there are a number of system-design issues about patterns that deserve further elaboration. For example, one might ask about the *utility* of patterns. It should be understood that

there is always a positive "feedback" of discovering patterns -they indicate what are concrete and typical design solutions in building ITSs. Once a pattern is discovered, it can be used at a proper place in another system/development/application, i.e. for further developments. Patterns also help in making an explicit and systematic classification of ITS design issues and bridge the gap between learning theories and experimentation on one side, and practical systems on the other side. Note, however, that different levels of abstraction of patterns imply different kinds of their utility. For example, higher-level patterns like GPA help in making more global architectural decisions, while lower-level ones (such as those that make up the core of PLAIT) can be useful tools in refining coarse-grained architectures.

Patterns in ITSs can appear at different levels of abstraction, which usually correspond to different levels of granularity as well. In previous sections we presented patterns related mostly to general architectural styles of ITSs or of some of their subsystems. Therefore they are called *architectural patterns* (Buschmann et al., 1996). But patterns can also be noted at a more concrete level: *design patterns* give typical solutions for the structure of subsystems (Gamma et al., 1995) and typically have only a few classes. The use of one of these patterns (the Composite pattern) for hierarchical organization of learning material in ITSs is shown in (Devedzic, 1999b). Sometimes it is difficult to specify the exact levels of abstraction and granularity for a pattern. While extracting the GPA pattern we found that the granularity of agents differs strongly in some agent-based ITSs. Some of the systems have agents with a lot of functionality and quite a complex inner structure (Rickel et al., 1999) while other systems use lots of agents, each with a very specific function (Chen et al., 1999). Perhaps it indicates that for an exact classification of patterns in terms of their abstraction and granularity we also need a common vocabulary of the concepts related to the patterns. That would result in an ontology for the structural elements of ITS used in patterns (Devedzic, 1999a).

As already mentioned in the section on the methodology we used for discovering the patterns described here, our knowledge of general software patterns was very useful in the analysis of ITS-related patterns we have discovered. As a rule, the lower the granularity or scale of a pattern is (e.g., simple design-level patterns as in (Gamma et al., 1995)), the more general and the less ITS-specific it is. The larger the granularity of a pattern is (e.g., analysis-level patterns) and thus the more specific components of an ITS it involves, the more chance that the patterns discovered will be ITS-specific. Note that *both* general and ITS-specific ones are useful, each in their own way (see above). As a good engineering practice, for each ITS-specific pattern discovered one should document which of the known general software patterns are related to it. Another good engineering practice suggests that ITS architectures should conform to general-purpose software patterns for the sake of good system engineering, but that they also must reflect ITS-specific issues. Some simple patterns may be of both natures, thus serving as links between the two categories of patterns.

These patterns and also the patterns that can be used perfectly outside the ITS field, nevertheless should not be neglected when compiling a pattern language. This comes from the claim for a pattern language to provide a certain "completeness" for the specific domain (Schmidt et al, 2000). If a general pattern is also useful in the context of ITSs, then it should be included into the ITS pattern language, because it can be used in combination with other patterns of the language.

Pattern discovery in software architectures often starts from analyzing and comparing topologies of different but functionally similar systems. The danger of that approach alone is the

possibility of coming up with too abstract patterns, for which it is hard to see a connection to real-world systems. Another common pitfall is that a pattern discovered only by topological analysis of existing systems may merely reflect a common abstract substructure that could not be called a pattern. In the case of ITS architectures, this is quite possible -the search space is fairly small (especially if it is constrained to some highly specific architectures), and with relatively little effort one might find out a common substructure that is unfortunately of little use to designers because it is too abstract and perhaps lacks more clear semantics. The best way to cope with these problems is to provide strong evidence of recurrence of such patterns, to clearly explain both the context in which the newly discovered pattern applies and what are believed to be the driving forces for the designers who used them, and to document the pattern with "known uses" (i.e. with the reference to actual systems that have used it). On the other hand, in the context of ITS architectures we have found the patterns' topology to be very indicative of the overall design correctness -the more a newly discovered ITS-architectural pattern resembles general-purpose software patterns, the more stable the ITS architecture that uses it.

However, topology is not everything. Architecturally, it *should* drive the development process for the sake of software engineering issues, but may be conceptually insufficient. ITS-specific issues may add flesh to the skeleton provided by topology (architecture).

Pattern names represent another important issue. For example, should a pattern name reflect just the pattern's topology, or other issues as well? How does one choose pattern names? How long should they be? How abstract? Should they better reflect domain metaphors, or should they reflect the system structure? And so on. The pattern's name conveys the essence of the pattern succinctly (Gamma et al., 1995). A good name is vital, because it will become a part of analysis and design vocabularies and will be used in communication between designers. We believe that it is essential for an ITS-architectural pattern name to be as descriptive as possible, but simultaneously compact and domain-specific whenever possible. Names of higher-level patterns should be rooted in domain metaphors, while names of patterns of lower-level, detailed design may come from the patterns' topology, due to the fact that lower-level patterns are less domain-specific anyway.

Because the vocabulary in the pattern community is quite young, misunderstandings about the meaning may arise. Especially since the knowledge transfer from software engineering into the AIED field has just begun in recent years, ambiguous or even ill defined use of terms can happen easily when the common vocabulary is not yet established. That can be seen in the very different granularity of agents in ITSs and also in discussions on whether the term "design pattern" encompasses also instructional design patterns, or should be used exclusively in the meaning of patterns of software design.

Patterns are a powerful technique for the design of software systems, because they propose solutions to typical problems of software engineering. But even if patterns are not used in the early phases of system development, the knowledge of patterns can help to keep the system easy to maintain and extend: architectural and design patterns can be introduced into an existing system's structure with a technique called *refactoring* (Fowler et al., 1999). This technique uses small-scale code transformations (such as extracting repeated code into a method, or defining abstract superclasses to allow polymorphism), that improve the structure of the code and also preserve the system's behaviour. With several of these refactoring steps, patterns may be introduced into a system after the system has been implemented. One can even refactor the system in such a way that it conforms to the interfaces of frameworks that provide concrete

classes, so not all the components have to be developed from scratch. An example for the use of this technique for educational systems can be found in (Harrer, 2002).

Another interesting question related to ITS architectures and patterns is that of constraints imposed by ITS authoring tools. When an ITS is developed using an authoring tool, a lot of flexibility and a number of possibilities are usually offered by the tool in terms of how the user interface, the domain expertise, the student model, the tutoring model, and the other parts of the ITS can be authored (Murray, 1999). In all such cases, the resulting system architecture is constrained by the ITS model(s) the authoring tool supports. A nice extension to authoring tools might be to make them support several reference architectures, preferably after patterns like *Classic ITS architecture* and *KnowledgeModel-View*. The user could then select a preferred pattern and get help on advantages and disadvantages of the reference architecture selected and on possible follow-up patterns to be used in specific parts of the ITS, following the relations within a pattern language. Likewise, it would be also possible to extend authoring tools in terms of providing pattern-based development of parts of the ITS (student model, tutoring model, and so on). An integration of pattern aspects in authoring tools would reduce the gap between authoring and development environments and give authored ITSs an architecture well prepared for future extensions and maintenance. However, further research and pattern discovery in all subfields of ITS is needed before such extensions become viable.

Yet another word of warning about patterns: whenever a new pattern is discovered, it should be evaluated by other designers and specialists in the field. It is a long and collective work to agree on a set of patterns in any domain. Such an external evaluation of the patterns described in this paper is currently underway. A positive feedback has already come from specialists in software architectural design; we now need feedback from ITS specialists.

RELATED WORK

Pattern-related issues and efforts are subject of a number of recent projects and efforts in the broad fields of ITS, e-Learning, and education in general. Such kinds of work span different subject areas. In addition to the work already mentioned in the Introduction (Katz et al., 1999; Inaba et al., 2001; Scott & Reif, 1999), researchers have already discovered patterns in educational processes and activities and in general educational design, pedagogical patterns, instructional design patterns, and learner interaction patterns in collaborative systems. All such patterns are extremely useful in designing and developing new learning technology and interactive learning environments.

For example, Centre for User-Oriented IT Design (CID) of the Department of Numerical Analysis and Computer Science (NADA) at the Royal Institute of Technology, Stockholm, Sweden (KTH) has published several general educational design patterns and shown instances of those patterns in present mathematics education (http://kmr.nada.kth.se/cm/edupatterns.html). Likewise, several instructional design patterns related to content literacy development are published at http://165.248.178.100/compact/patterns/patterns.html. They cover important issues of instructional design in the field, such as language development preliteracy readiness, language-based word recognition, comprehension focus, word recognition focus, meaning-based word recognition, and direct skills approach to word recognition.

One of the objectives of the e-LEN project (http://www2.tisip.no/E-LEN/) is to maintain a repository of design patterns for e-Learning. There is already a couple of dozen such patterns in the repository, divided into four categories: patterns for learning resources and learning management systems, patterns related to lifelong learning, patterns for collaborative learning, and patterns in the domain of adaptive learning. Newly discovered patterns can be contributed to the repository.

The PoInter project (http://www.comp.lancs.ac.uk/computing/research/cseg/projects/pointer/) is concerned with investigating the appropriateness of patterns as a means of communicating information about how people interact with each other through and around technology. The aim of the TELL project (http://elessar.ted.unipi.gr/en/projects/TELL) is to identify design patterns for effective network-supported collaborative learning. Yet another important recent initiative is the Pedagogical Patterns Project (http://www.pedagogicalpatterns.org). This project deals with patterns that are designed to capture expert knowledge and best practices of teaching and learning in a specific domain. Pedagogical patterns try to capture expert knowledge of the practice of teaching and learning. The pedagogical patterns discovered and published by this project consortium describe the essence of the best practices in a compact form that can be easily communicated to those who need the knowledge, and represent a pattern language of reusable pedagogical design patterns.

Along with the projects and initiatives, research publications on patterns for ITS and e-Learning started to pop up here and there. Early work, such as (Devedzic, 1999a; Devedzic, 1999b; Mizoguchi & Bourdeau, 2000), remained rather isolated for a while. More recently, Frizell and Hübscher (2002a; 2002b) presented a methodology for supporting the use of patterns during Web-based instructional design. This methodology consists of a pattern language for Web-based instruction and a design environment that scaffolds the process of finding, selecting, and applying patterns to design problems. Rodriguez et al. (2004) have developed a pattern language with proper support for learning design, relating it to constructs in the current IMS-LD specification. Hernández Leo et al. (2004) have also used IMS-LD specification as the basis for formalizing collaborative learning patterns and generating an appropriate educational modeling language. Avgeriou et al. (2003) have proposed a pattern language for designing learning management systems, and Frosch-Wilke (2004) used proven design patterns from object-oriented software design for modeling different metadata elements of learning objects and their relationships. Another interesting recent result in the domain of capturing Web-based learning processes and their subsequent instantiation on learning technology in the form of reusable patterns is CEWebS, an open Web-service-based learning technology architecture designed for supporting the implementation of these patterns with conceptual guidance by the layered Blended Learning Systems Structure (BLESS) model (Derntl & Mangler, 2004).

There is also a working ITS for teaching design patterns frequently used in software engineering. It is called DP-ITS (Jeremic et al., 2004; Jeremic & Devedzic, 2004). It covers the patterns from the most authoritative book on design patterns to date (Gamma et al., 1995), and has a sophisticated student model that accounts not only for the student's performance, but also for his/her cognitive traits.

CONCLUSIONS

Discovering patterns in ITS-related issues like ITS architectures is not a kind of search for *new* modelling and design solutions. It is more like *compiling* what is already done in practice. Patterns enable us to see what kind of solutions ITS/AIED researchers and developers *typically* apply when faced with common problems. In the future, we expect discovery of many useful patterns other than those that we have discovered so far. The efforts in describing them in an appropriate way will follow, finally resulting in accumulation of the patterns in catalogues and repositories over time. This way the efforts will actually unveil common structures of frequently arising problems and will help describe and represent the knowledge of their contexts, solutions, driving forces and trade-offs in the form of explicit statements. Once we have explicitly represented such knowledge and experience, we can get valuable feedback -we can use the patterns intentionally and systematically in other systems and applications, i.e. for further developments. In that sense, AIED/ITS patterns can be understood as many small tools for modelling and developing ITSs, and AIED/ITS pattern languages, pattern catalogues, and repositories as the corresponding toolkits.

As a concrete direction for our future research, we intend to study the architectures of different ITS authoring tools for possible discovery of new patterns. The earlier work of Murray (1999) indicates that architectures of different ITS authoring tools may have much in common, i.e. that such architectures may contain a number of recurring structures at different levels (such as the supported learner model, tutoring model, and the like). In fact, a similar approach has been successfully taken by Avgeriou et al. (2003) -they were mining for patterns in more than a dozen different commercial learning management systems such as WebCT, Blackboard, and TopClass.

Many open issues about ITS/AIED patterns still remain and need to be further investigated. For example, do cognitive processes really work the way the discovered patterns (especially for GPA-agents) suggest? If so, to what extent? What kinds of interaction patterns exist among pedagogical agents in multiagent educational systems? Do they exist in the same way among human tutors, learners, peers, and assistants in different educational settings? In other words, further research on "cognitive justification" of patterns is necessary. However, we must not forget the fact that ITSs are a kind of *software system*; hence well-established practices of software analysis and design *do* matter. In that sense, we must recognize that knowledge of patterns lets us design our systems better.

REFERENCES

Aïmeur, E., Dufort, H., Leibu, D., & Frasson, C. (1997). Some Justifications for the Learning by Disturbing Strategy. In B. du Boulay, & R. Mizoguchi (Eds.) *Artificial Intelligence in Education* (pp. 119-126). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Aïmeur, E., & Frasson, C. (1996). Analyzing a New Learning Strategy according to different knowledge levels. *Computer and Education,* 27, 115-127.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). A Pattern Language -Towns Buildings Construction, Vol. 2 of Center for Environmental Structure Series. New York: Oxford University Press.

Alexander, C., Ishikawa, S., & Silverstein, M. (1979). The Timeless Way of Building, Vol. 1 of Center for Environmental Structure Series. New York: Oxford University Press.

Avgeriou, P., Papasalouros, A., Retalis, S., & Skordalakis, E. (2003). Towards a Pattern Language for Learning Management Systems. *Educational Technology & Society*, 6, 2, 11-24. [Online]. Available: http://ifets.ieee.org/periodical/6-2/2.html.

Beck, J., Stern, M., & Woolf, B.P. (1997). Cooperative Student Models. In B. du Boulay, & R. Mizoguchi (Eds.) *Artificial Intelligence in Education* (pp. 127-134). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Brophy, S., Biswas, G., Katzlberger, T., Bransford, J., & Schwartz, D. (1999). Teachable Agents: Combining Insights from Learning Theory and Computer Science. In S.P. Lajoie, & M. Vivet (Eds.) *Artificial Intelligence in Education* (pp. 21-28). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Brusilovsky, P. (1995). Intelligent learning environments for programming: The case for integration and adaption. In J. Greer (Ed.) *Proceedings of the World Conference on Artificial Intelligence in Education AI-ED 95* (pp. 1-7). Charlottesville, VA: AACE.

Brusilovsky, P. (2003). A Component-Based Distributed Architecture for adaptive Web-based Education. In U. Hoppe, F. Verdejo, & J. Kay (Eds.) *Artificial Intelligence in Education* (pp. 386-388). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Bull, S., Brna, P., Critchley, S., Davie, K., & Holzherr, C. (1999). The Missing Peer, Artificial Peers and the Enhancement of Human-Human Collaborative Student Modeling. In S.P. Lajoie, & M. Vivet (Eds.) *Artificial Intelligence in Education* (pp. 269-276). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *A System of Patterns*. Chichester: John Wiley & Sons.

Canut, M., Guarderes, G., & Sanchis, E. (1999). The Systemion: A New Agent Model to Design Intelligent Tutoring System. In S.P. Lajoie, & M. Vivet (Eds.) *Artificial Intelligence in Education* (pp. 54-63). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Chan, T.-W., & Baskin, A.B. (1988). Studying with the Prince -The Computer as a Learning Companion. *Proceedings of the First International Conference on Intelligent Tutoring Systems, ITS'88* (pp. 194-200). Montreal, Canada.

Chan, T.-W., & Chou, C.-Y. (1997). Exploring the Design of Computer Supports for Reciprocal Tutoring. *International Journal of Artificial Intelligence in Education,* 8, 1-29.

Chen, W., & Mizoguchi, R. (1999). Communication Content Ontology for Learner Model Agent in Multi-Agent architecture. In G. Cumming, T. Okamoto, & L. Gomez (Eds.) *Proceedings of ICCE '99, 7th International Conference on Computers in Education* (pp. 95-102). Kuching, Japan, Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Chou, C.-Y., Lin, C.-J., & Chan, T.-W. (1999). User Modeling in Simulated Learning Companions. In S.P. Lajoie, & M. Vivet (Eds.) *Artificial Intelligence in Education* (pp. 277-284). Amsterdam: IOS Press/ Tokyo: OHM Ohmsha.

Coplien, J.O., & Zhao, L. (2005). *Toward a General Formal Foundation of Design - Symmetry and Broken Symmetry*. Brussels: VUB Press.

Crowley, R., Medvedeva, O., & Jukic, D. (2003). SlideTutor: A model-tracing Intelligent Tutoring System for teaching microscopic diagnosis. In U. Hoppe, F. Verdejo, & J. Kay (Eds.) *Artificial Intelligence in Education* (pp. 157-164). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Devedzic, V. (1999a). Ontologies: Borrowing from Software Patterns. *ACM Intelligence Magazine* 10/3, 14-24.

Devedzic, V. (1999b). Using design patterns in ITS development. In S.P. Lajoie, & M. Vivet (Eds.) *Artificial Intelligence in Education* (pp. 657-659). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Devedzic, V. (2001). A Pattern Language for Architectures of Intelligent Tutors. *IEEE Learning Technology Newsletter* 3/3, available at: http://lttf.ieee.org/learn_tech/issues/july2001/index.html#8.

Devedzic, V. (2002). Software Patterns. In S.K. Chang (Ed.) *Handbook of Software Engineering and Knowledge Engineering Vol.2 -Emerging Technologies* (pp. 645-671). Singapore: World Scientific Publishing Co.

Derntl, M., & Mangler, J. (2004). Web Services for Blended Learning Patterns. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies, ICALT'04* (pp. 614-619). Los Alamitos: IEEE Press.

Dillenbourg, P., Jermann, P., Schneider, D., Traum, D., & Buiu, C. (1997). The design of MOO agents: Implications from an empirical CSCW study. In B. du Boulay, & R. Mizoguchi (Eds.) *Artificial Intelligence in Education* (pp. 15-22). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Farance, F., & Tonkel, J. (2001). Learning technology systems architecture. *Architecture Specification Unapproved Draft 9*, IEEE Learning Technology Standards Committee.

Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Reading, MA: Addison-Wesley.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.

Frasson, C., Mengelle, T., & Aimeur, E. (1997). Using Pedagogical Agents in a Multi-Strategic Intelligent Tutoring System. In Proceedings of the Pedagogical Agents Workshop (pp. 40-47), The 8th World Conference on Artificial Intelligence in Education, AIED'97, Kobe, Japan, August 18-22.

Frasson, C., Mengelle, T., Aimeur, E., & Guarderes, G. (1996). An Actor-Based Architecture for Intelligent Tutoring Systems. In *Proceedings of the Third International Conference on Intelligent Tutoring Systems, ITS'96* (pp. 57-65). Montreal, Canada.

Frizell, S.S., & Hübscher, R. (2002a). Supporting the Application of Design Patterns in Web-Course Design. In *Proceedings of ED-MEDIA 2002*, Denver. AACE.

Frizell, S.S., & Hübscher, R. (2002b). Aligning Theory and Web-based Instructional Design Practice with Design Patterns. In *Proceedings of E-Learn-World Conference on E-Learning in Corporate, Government, Healthcare & Higher Education*. Montreal, 2002.

Frosch-Wilke, D. (2004). An Extended and Adaptable Information Model for Learning Objects. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies, ICALT'04* (pp. 166-170). Los Alamitos: IEEE Press.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Gonschorek, M. (1998). Wissensbasierte Integration und Adaption von Praesentationen in Intelligenten Lehrsystemen. *Ph.D. Thesis*. Munchen: Technische Universitaet Muenchen, Institut fuer Informatik.

Goodman, B., Soller, A., Linton, F., & Gaimari, R. (1998). Encouraging Student Reflection and Articulation Using a Learning Companion. *International Journal of Artificial Intelligence in Education,* 9, 237-255.

Hamburger, H., & Tecuci, G. (1998). Toward a Unification of Human-Computer Learning and Tutoring. In B.R. Goettl, H.M. Halff, C.L. Redfield, & V.J. Shute (Eds.) *Intelligent Tutoring Systems, Lecture Notes in Computer Science 1452* (pp. 444-453). New York: Springer-Verlag.

Harrer, A. (2000). Unterstuetzung von Lerngemeinschaften in verteilten intellgenten Lehrsystemen. *Ph.D. Thesis*. Munchen: Technische Universitaet Muenchen, Institut fuer Informatik.

Harrer, A. (2001). Analysis of social interaction for construction of group models. In J. Moore, C. Redfield, & W.L. Johnson, (Eds.) *Proc. of the World Conference on Artificial Intelligence in Education AI-ED 2001 -AI-ED in the Wired and Wireless Future* (pp. 554-556). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Harrer, A. (2002). Software engineering methods for the re-use of existing components in educational systems. In *Proceedings of Applied Informatics (AI 2002)*, CD edition. Calgary: Acta Press.

Harrer, A., & Herzog, C. (1999). SYPROS going IDLE -from a classical ITS to an intelligent distributed learning environment. In G. Cumming, T. Okamoto, & L. Gomez (Eds.) *Proceedings of ICCE '99,*

7th International Conference on Computers in Education (pp. 836-839). Kuching, Japan, Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Hernández Leo, D., Asensio Pérez, J.I., & Dimitriadis, Y.A. (2004). IMS Learning Design Support for the Formalization of Collaborative Learning Patterns. In *Proceedings of The 4th IEEE International Conference on Advanced Learning Technologies, ICALT 2004* (pp. 350-355). Los Alamitos, CA: IEEE Press.

Hietala, P., & Niemirepo, T. (1998). The Competence of Learning Companion Agents. *International Journal of Artificial Intelligence in Education,* 9, 178-192.

Inaba, A., Ohkubo, R., Ikeda, M., Mizoguchi, R., & Toyoda, J. (2001). An Instructional Design Support Environment for CSCL -Fundamental Concepts and Design Patterns. In J. Moore, C. Redfield, & W.L. Johnson (Eds.) *Proc. of the World Conference on Artificial Intelligence in Education AI-ED 2001 -AI-ED in the Wired and Wireless Future* (pp. 130-141). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Jeremic, Z., Devedzic, V., & Gasevic, D. (2004). An Intelligent Tutoring System for Learning Design Patterns. In *Proceedings of the Workshop on Adaptive Hypermedia and Collaborative Web-based Systems (AHCW'04),* held in conjunction with the 4th International Conference on Web Engineering, Munich, Germany, July 26-30, 2004. [Online]. Available: http://www.ii.uam.es/%7Ercarro/AHCW04/Jeremic.pdf.

Jeremic, Z., & Devedzic, V. (2004). Design Pattern ITS: Student Model Implementation. In *Proceedings of The 4th IEEE International Conference on Advanced Learning Technologies, ICALT 2004* (pp. 864-865). Los Alamitos, CA: IEEE Press.

Johnson, W.L., Rickel, J., & Lester, J.C. (2000). Animated Pedagogical Agents: Face-to-Face Interaction in Interactive Learning Environments. *International Journal of Artificial Intelligence in Education*, 11, 47-78.

Johnson, W.L., Kole, S., Shaw, E., & Pain, H. (2003). Socially Intelligent Learner-Agent Interaction Tactics. In U. Hoppe, F. Verdejo, & J. Kay (Eds.) *Artificial Intelligence in Education* (pp. 431-433). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Katz, S., Aronis, J., & Creitz, C. (1999). Modeling Pedagogical Interactions with Machine Learning. In S.P. Lajoie, & M. Vivet, (Eds.) *Artificial Intelligence in Education* (pp. 543-550). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Koedinger, K.R., Anderson, J.R., Hadley, W.H., & Mark, M.A. (1997). Intelligent Tutoring Goes to School in the Big City. *International Journal of Artificial Intelligence in Education*, 8, 30-43.

Lester, J.C., Towns, S.G., & Fitzgerald, P.J. (1999). Achieving Affective Impact: Visual Emotive Communication in Lifelike Pedagogical Agents. *International Journal of Artificial Intelligence in Education,* 10, 278-291.

Marsella, S., Johnson, W.L., & LaBore, C. (2003). Interactive Pedagogical Drama for Health Interventions. In U. Hoppe, F. Verdejo, & J. Kay (Eds.) *Artificial Intelligence in Education* (pp. 341-348). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Mizoguchi, R., & Bourdeau, J. (2000). Using Ontological Engineering to Overcome Common AI-ED Problems. *International Journal of Artificial Intelligence in Education,* 11, 107-121.

Mitrovic, A. (2003). An Intelligent SQL Tutor on the Web. *International Journal of Artificial Intelligence in Education,* 13 (2-4), 173-197.

Murray, T. (1999). Authoring Intelligent Tutoring Systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education,* 10, 98-129.

Paiva, A. (1997). Learner modelling for collaborative learning environments. In B. du Boulay, & R. Mizoguchi (Eds.) *Artificial Intelligence in Education* (pp. 215-222). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Paiva, A., & Machado, I. (1998). Vincent, an Autonomous Pedagogical Agent for On-the-Job Training. In B.R. Goettl, H.M. Halff, C.L. Redfield, & V.J. Shute (Eds.) *Intelligent Tutoring Systems, Lecture Notes in Computer Science 1452* (pp. 584-593). New York: Springer-Verlag.

Rickel, J., & Johnson, W.L. (1999). Virtual Humans for Team Training in Virtual Reality. In S.P. Lajoie, & M. Vivet, (Eds.) *Artificial Intelligence in Education* (pp. 578-585). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Rodríguez, M.C., Rifón, L.A., & Nistal, M.L. (2004). Towards IMS-LD Extensions to Actually Support Heterogeneous Learning Designs. A Pattern-based Approach. In *Proceedings of The 4th IEEE International Conference on Advanced Learning Technologies, ICALT 2004* (pp. 565-569). Los Alamitos, CA: IEEE Press.

Russell, S. & Norvig, P. (2002). *Artificial Intelligence -A Modern Approach* (2nd Ed.). Englewood Cliffs, NJ: Prentice-Hall.

Schmidt, D., Fayad, M., & Johnson, R.E. (1996). Software Patterns. *Communications of The ACM,* 39/10, 37-39.

Schmidt, D., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-oriented Software Architecture - Patterns for Concurrent and Networked Objects*. Chichester: John Wiley & Sons.

Scott, L.A., & Reif, F. (1999). Teaching Scientific Thinking Skills: Students and Computers Coaching Each Other. In S.P. Lajoie, & M. Vivet, (Eds.) *Artificial Intelligence in Education* (pp. 285-293). Amsterdam: IOS Press/Tokyo: OHM Ohmsha.

Shaw, E., Ganeshan, R., Johnson, W.L., & Millar, D. (1999). Building a Case for Agent-Assisted Learning as a Catalyst for Curriculum Reform in Medical Education. In *Proceedings of the Workshop "Animated and Personified Pedagogical Agents"* (pp. 70-78). Le Mans, France.

Shimic, G., & Devedzic, V. (2003). Building an intelligent system using modern Internet technologies. *Expert Systems With Applications*, 25, 231-246.

Suthers, D. (2001). Architectures for Computer Supported Collaborative Learning. In T. Okamoto, R. Hartley, Kinshuk, J.P. Klus (Eds.) Proceedings IEEE International Conference on Advanced Learning Technology: Issues, Achievements and Challenges (pp. 25-28). Los Alamitos, CA., IEEE Computer Society.

Tecuci, G., & Keeling, H. (1999). Developing Intelligent Educational Agents with Disciple. *International Journal of Artificial Intelligence in Education*, 10, 221-237.

Vassileva, J. (1998). Goal-Based Autonomous Social Agents: Supporting Adaptation and Teaching in a Distributed Environment. In B.R. Goettl, H.M. Halff, C.L. Redfield, & V.J. Shute (Eds.) *Intelligent Tutoring Systems, Lecture Notes in Computer Science 1452* (pp. 564-573). New York: Springer-Verlag.

Wang, W.-C., & Chan, T.-W. (2000). CAROL5: An Agent-Oriented Programming Language for Developing Social Learning System. *International Journal of Artificial Intelligence in Education,* 11, 1-39.

Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems*. Los Altos, CA: Morgan Kaufmann.

Winn, T., & Calder, P. (2002). Is This a Pattern? *IEEE Software*, 19/2, 59-66.

Winter, M., & McCall, G. (2003). An Analysis of Group Performance in Terms of the Functional Knowledge and Teamwork Skills of Group Members. In U. Hoppe, F. Verdejo, & J. Kay (Eds.) *Artificial Intelligence in Education* (pp. 261-268). Amsterdam: IOS Press / Tokyo: OHM Ohmsha.

Yin, J., El-Nasr, S., Yang, L., & Yen, J. (1998). Incorporating Personality into a Multi-Agent Intelligent System for Training Teachers, In B.R. Goettl, H.M. Halff, C.L. Redfield, & V.J. Shute (Eds.) *Intelligent Tutoring Systems, Lecture Notes in Computer Science 1452* (pp. 484-493). New York: Springer-Verlag.