Monograph: Engineering e-Learning Systems

(As published jointly with Novática*)

Guest Editors: Ignacio Aedo-Cuevas, Kinshuk, Demetrios G. Sampson, Antonio Sarasa-Cabezuelo, and José-Luis Sierra-Rodriguez


3. A Pervasive Learning Design Methodology — Jorge Llácer-Gil de Ramales

4. Communication Patterns in Component-Based Intelligent Tutoring Systems — Géraldine Ruddeck, Dennis Maciuszek, Martina Weicht, and Alke Martens

5. Intelligent Tutoring Systems with SCORM — Gustavo Soares Santos and Álvaro Reis Figueira

6. PDP Systems for Audiovisual Educational Content: Development of a Solution for the Universidad de La Laguna — Carina-Soledad González-González, David Cabrera-Primo, Daniel López-Barrionuevo, and Antonio Barroso-Diaz

7. Component Reuse in Synchronous Collaborative Web 2.0 Applications — Tiago Caminha Gaspar, Cesar Augusto Camilo Teixeira, and Antonio Francisco do Prado


9. Selected CEPIS News — Fiona Fanning

* This monograph will be also published in Spanish (full version printed; summary, abstracts, and some articles online) by Novática, journal of the Spanish CEPIS society ATI (Asociación de Técnicos de Informática, <http://www.ati.es>), and its journal Novática.
Design patterns are reusable solutions to recurrent problems. Generally described in the context of object-oriented programming, only a few contributions can be found in the domain of eLearning in general or Intelligent Tutoring Systems (ITS) in particular. Based on several approaches to describe the systems’ underlying architecture in terms of pattern collections, we have developed a component-based framework. This framework provides a foundation for the development of flexible and reusable ITS. It also provides a basis for the development of other eLearning systems. Still, the complexity of the inter-modular communication requires experience and self-discipline from the programmer. Thus we investigated the framework for additional patterns. We extracted useful communication patterns and integrated them into an editor. Patterns and editor are presented in this paper as they have already proved themselves very useful regarding programming efficiency and consistency.

**Keywords:** Component Pattern, Framework, Intelligent Tutoring System, Software Engineering.

**1 Introduction**

Having their roots in architecture [1], patterns have been adopted by Computer Science and software development since the late 1980s [2]. Since the early 2000s, researchers in the domain of eLearning have made attempts to describe their systems using patterns. The first approaches can be traced back to Devedzic, who collected and described patterns for Intelligent Tutoring Systems, ITS, in 1999 [3]. Harrer and Martens have extended this work towards a catalogue of patterns for ITS (see e.g. [4][5]). Martens and Cap [6] showed that patterns can be used to describe eLearning systems in general, and Maciuszek and Martens extended the pattern catalogue with regard to game patterns [7] in the domain of game-based learning.

Generally, the pattern catalogue started in [5] is based on the idea of combining patterns at different levels of abstraction and integrating them into one collection. It contains ITS-specific patterns and general eLearning patterns, and software design patterns based on the work of Gamma et al. [2]. The catalogue aims at providing a collection of descriptions to be re-used at different levels of project development. While the core patterns of the ITS can easily be used in project structuring and inter-team communication [8], the detailed patterns are best suited for domain expert communication. The related software design patterns support the implementation phase of an eLearning system.

But why do we continue to develop patterns, and why do we think this is a valuable approach? ITS research can look back on a long tradition, and the resulting systems are usually based on more or less the same basic elements. These basic elements can be traced back to a description by Clancy in the 1980s [9]. He noted that, although the names vary, ITS mainly consist of four "models", i.e. user interface, user model, expert knowledge model, and pedagogical knowledge model. However, the drawback of this description is that role and functionality of each of the models are com-

**Authors**

Géraldine Ruddeck is a PhD student at the University of Rostock, Germany, in the domain of eLearning and Cognitive Systems. She is part of the Research Training Group "Integrative Development of Modeling and Simulation Methods for Regenerative Systems" (DIEM oSiRiS), an interdisciplinary research group involving Biologists and Computer Scientists. Her research interests include game-based learning and computational thinking. She has contributed to a number of media design projects, both as a freelancer and as a student at Stuttgart Media University, Germany. <geraldine.ruddeck@uni-rostock.de>

Dennis Maciuszek is a PhD student at the University of Rostock, Germany, Institute of Computer Science, eLearning and Cognitive Systems group, where he is doing research in the area of Game-based Learning. He holds a German Diploma and a Swedish Licentiate of Engineering in Computer Science (minor: Psychology), as well as a Master of Arts in Media. He has previously been employed at the Technical University of Braunschweig, Germany, and Linköpings Universitet, Sweden, working in Software Design, eLearning/Intelligent Tutoring Systems, and Assistive Technology. In addition, he has contributed to a number of media design projects, both as a freelancer and as a student at Stuttgart Media University, Germany. <dennis.maciuszek@uni-rostock.de>

Martina Weicht is a PhD student at the University of Rostock, Germany, Institute of Computer Science, eLearning and Cognitive Systems group. She is currently working in an EU project on Virtual Laboratories, researching in Intelligent Tutoring Systems with a special interest in Inclusive eLearning. <martina.weicht@uni-rostock.de>

Alke Martens is Professor for eLearning and Cognitive Systems at the University of Rostock, Germany, Institute of Computer Science. Her research interests include game-based learning, adaptation technologies for teaching and training systems, cognitive models of users, ubiquitous and pervasive learning and system engineering. She has several projects, is chair and co-chair of several international conferences and workshops, and is editor-in-chief of a German journal for Instruction in Computer Science. <alke.martens@uni-rostock.de>
This article emphasizes the importance of communication in component-based software engineering, specifically in the context of Intelligent Tutoring Systems or ITS. Completely unclear, leading to a situation where ITS architectures are not really comparable, where each ITS developer "re-invents the wheel" and where neither parts of existing software nor main ideas underlying the implementation or design of an ITS can be re-used (see e.g. [5][10]). The same situation (and even worse due to the large amount of different system types) can be observed in eLearning development.

To demonstrate the usability of our patterns, we used them in the development of a component-based framework [11] called JaBInT [10]. In the following we will use the terms "semantic component", "module" and "framework", which will be explained briefly in Section 2.

The idea underlying the JaBInT framework is called "plug-n-train", as modules can be plugged together to develop or extend the functionalities of an ITS.

In the following, we will sketch the framework before describing the communication patterns which we have detected in use of the framework. These patterns facilitate the development and re-use of modules (i.e. the "plugs") for eLearning systems developed with our framework. They have been integrated into an authoring system for ITS, allowing the user to very easily "click 'n plug" an ITS from the defined patterns.

2 The JaBInT Framework for ITS

The term "framework" is used here in the software engineering sense: a framework is a set of re-usable structures which contain the main functionality. They are implemented as concrete and abstract classes defining their basic functionality. New components must implement the interfaces to extend the basic functionality. Software components are part of a composition (a software system or a framework) and can be deployed independently by other software developers. The single component is usually developed based on a component model, describing the basic structure of the component, its interfaces and the communication structure. A well-known example of a component based approach is the Eclipse framework, <http://www.eclipse.org>. Just like Eclipse, JaBInT is developed as a framework – in contrast to other approaches, e.g. service oriented architectures.

The JaBInT framework defines four "semantic components": User Interface, User, Expert and Process Steering. The actual functionality of the ITS is implemented in modules, which can communicate via ports and communication channels. Each module is assigned to a semantic component based on the aim of its functionality, e.g. a user history or user preferences are assigned to the User semantic component. Module assignment to a semantic component and inter-module communication are provided by JaBInT. On the design level, semantic components can be seen as containers for modules. They have no functionality themselves but serve the system developer(s) to clarify the structure of the ITS and the repartition of its functionalities. On the implementation level, the semantic components are only mentioned in an XML, Extensible Markup Language, file describing the structure of the ITS, while modules are packages possibly containing several classes, simulators, databases etc.

The real functionality of the ITS is contained within the modules. They communicate with each other along connections joining output ports to input ports. The input ports can be either "master" or "slave", which means that an incoming message will either "wake up" the module or will wait to be read. The input ports and output ports are the only interface between the modules. Communication in our sense is a unidirectional logical channel between a specific output port and a specific input port, defined in an XML configuration file. It can "carry" a message, which in Java is an instance of a specific message class. We may, for example, define a class NewTaskRequest containing three variables CurrentTaskID, CurrentTaskDifficulty and UserID along with their corresponding "get" and "set" methods. A NewTaskRequest object with all the variables set to the appropriate value can be sent via a connection between two modules from the User Interface to the Process Steering semantic component. We would do so to carry the information about a current task and user in order to receive a new task in return. This way, when replacing a module by another, the developer only needs to ensure that the ports and messages are the same in both modules. Otherwise the modules are completely independent from each other.

This is indeed a very flexible framework, but it also provides the programmer with far more possibilities than would make sense. For instance, one could misuse a communication channel between two ports to convey different message types. Another drawback of the framework lies in the management of the ports and communications: Implementing a new communication between two modules A and B is a rather complex task, as it is necessary to define:

- an output port for module A
- an input port for module B
- the input port type ("master" or "slave")
Figure 1: Implementation Level View on the Structure of an ITS implemented in JaBInT.

- a message class and
- a logical connection between the two created ports.

This is time-consuming and represents a considerable source of errors. Inconsistencies in the naming of ports, connections and messages (i.e. use of a different name for the input port, the output port and the message) do not constitute a real error but are strongly to avoided, as they make the implementation much harder. At the implementation level, the structure is slightly more complex than described on the design level. JaBInT holds the general structure of modules and communications in an XML file. Modules can be composed of several Java classes, some external resources, databases etc., and each module needs its own XML file to define its ports. Each message between two ports needs to be implemented in its respective Java class. The precise structure of an ITS implemented with JaBInT is illustrated in Figure 1.

Both to avoid misuse of the possibilities of JaBInT and to provide a more usable and secure framework, we defined some useful patterns regarding the inter-module communication and implemented them into an editor. These steps originate from our practical experience with JaBInT and provide modularity-friendly solutions to recurrent situations. They also limit some undesirable possibilities offered by the framework and spare the programmer much work.

3 Communication Patterns in Component-Based ITS

We define as communication the process of sending a message over a connection between an output port and a slave or master input port (we will speak of slave or master connection). Therefore communication patterns describe a structure of ports and connections enabling this communication.

3.1 Atomic Patterns

We define as "atomic pattern" a communication pattern containing a minimum of communications, so that all communication flows can be described as a combination of instances from the set of atomic patterns. From all the possible combinations involving two modules, we identified two which define a sufficient vocabulary for most issues:

- The first is the request pattern. It is used when module A needs information or any job done from module B. Module A sends a request message to a master port of module B and waits for the answer. Module B "wakes up" by the message, executes the order and sends the answer to a slave port of module A. The structure is very basic: a master connection from A to B and a slave connection from B to A (see Figure 2a top). In order to distinguish connection types, connections to master ports are represented with a plain arrow and connections to slave ports with a dashed arrow.

- The second and even more basic communication pattern is the instruction pattern. It is used when module A requires module B to perform a task without the need for feedback or simply transmits information for storage. Module A sends an instruction message to a master port of module B, which "wakes up" and executes the order or saves the information. This structure is only composed of a master connection from A to B (see Figure 2a bottom). In order to distinguish connection types, connections to master ports are represented with a plain arrow and connections to slave ports with a dashed arrow.

"A framework is a set of re-usable structures which contain the main functionality"
much more convenient to deal with request and instruction communications than with mere master and slave communications. We therefore declare these atomic patterns as a suitable and sufficient basic vocabulary for the development of ITS with JaBiT and will use them in the following for our work.

Another benefit of the exclusive use of the two atomic patterns is that cyclic connection pathways are avoided. Cycles increase the interdependency between modules and consequently decrease the simplicity of exchanging a single module. It is preferable to limit the connections to requests and instructions, even if it increases the total amount of connections.

3.2 Composed Patterns

Some features can be found in any ITS and are therefore considered worthwhile to be described in detail. These features, once formalized in pattern form, can then be used for the development of further ITS. This increases the flexibility of the resulting system while decreasing its implementation costs.

One of these recurrent situations is when the learner is required to complete a task and, in order to do so, performs several actions like clicking or writing. We name every such action an "entry" from the learner. The ITS may want to check whether the entry is correct and, if necessary, provide appropriate feedback.

![Figure 2](image-url)
Usually many steps are required to check whether a user entry is correct and to retrieve a feedback for the learner. First, the module $UI$ from the $UserInterface$ semantic component sends the data about the entry on a master input port of the appropriate module $P$ to the $ProcessSteering$ semantic component and waits for an answer (request communication). As checking every single entry may be too expensive, $ProcessSteering$ should comprise a module $Pc$ computing the necessity of a correctness checking. If the necessity is established, $P$ proceeds to the actual check by requesting the respective expert knowledge from the appropriate module $E$ of the $Expert$ semantic component and eventually decides about the correctness and returns an answer to $UI$.

In order to give feedback, the module $P$ needs two sorts of information: pedagogical rules from the didactical module $Pd$ of the $ProcessSteering$ semantic component, and information about the user (error history, personal preferences or learning type) from the module $U$ of the $User$ semantic component. These were ignored in the first step of our pattern definition but can be defined separately and then merged.

We may now combine both patterns. Considering that the learner’s errors should be stored, another connection needs to be added from the $P$ module to the $Ud$ module, where the error is processed and stored. Since this task requires no answer, it is a typical example for instruction connection usage. The resulting global pattern representing all ports and communication is shown in Figure 2b.

Thus, we saw a simple example of a very typical situation described as a pattern. Patterns can be combined as needed to adapt to a particular situation, allowing the implementer to consider the whole ITS communication at several abstraction levels, e.g. a “check and feedback” pattern composed of “entry and check”, “compute feedback” and “save user entry” pattern, implemented using the instruction and request atomic patterns.

4 Integrating the Patterns into an Editor

To simplify JaBlInT handling, we implemented an editor which provides an overview of the structure of the ITS including its semantic components, modules, ports and connections, and permits editing of this structure using the defined atomic and composed patterns. A new connection may be added easily on the design level by assigning a name and choosing the type of connection (request or instruction). This simplifies the process of adding new connections by automating the naming of ports and messages, and obliges the developer to think about the aim of the new connection in terms of expected answer (for a request) or independent information or command (for an instruction). This approach simplifies the complex process of adding a new connection to the essential component (providing a root name and a type), and ensures a systematic naming of all involved elements.

The editor (see Figure 3a) abstracts the implementation level and provides an overview of the structure of the ITS at the design level. It shows all modules sorted by semantic component in the upper part of the window. Ports (left) and connections (right) of the selected module are listed in the

Figure 3: (a) Screenshot of the JaBlInT editor; (b) Screenshot of the Flow Visualization in the Editor (the flows are listed at the top left box; the rest of the window shows the involved connections and the steps of the communication sequence).
middle. Selecting a connection highlights the corresponding ports and vice versa. At the bottom, a console displays the latest connections selected by the user along with their corresponding modules. A dialog for creating a new connection popped up on the left.

Since only two possibilities for creating a new connection are provided, the editor grants the exclusive use of the two atomic patterns (see Figure 3a, bottom left). Selecting the request pattern deactivates the "master" and "slave" radio buttons, since these only make sense for the instruction pattern. We chose to keep a certain flexibility in the editor rather than to hinder ourselves by our own pattern restriction in case the situation comes up that a slave instruction is inevitably required. If this situation never occurs in the course of our work with JaBiNT, the "master" and "slave" radio buttons will be erased from the dialogue window.

As the number of connections in an ITS implemented with JaBiNT increases quickly, it is important to keep an overview of communication sequences (called "flows") that take place in different situations. These sequences are partly built on predefined patterns, but may also be combinations of existing patterns, modified versions of a pattern or completely new sequences. The editor proposes a visualization of the connections and an interactive visualization of communication flows (see Figure 3b).

5 Special-Purpose Patterns
The patterns discussed so far document very general communication processes needed in any component-based ITS. The next step is to help developers working on more specialized applications. These could be ITS following a certain instructional approach, e.g. ITS based on collaborative inquiry learning, or game-based ITS that merge adaptive training with digital games [12]. Adaptive educational games promise to support intrinsic motivation, free experimentation, and illustrative simulation, adding personalized support. In our work, we apply a further differentiation between game-based ITS belonging to certain genres. We are working on ITS based on computer role-playing games.

"The idea underlying the JaBiNT framework is called "plug-n-train""
Communication patterns describe a structure of ports and connections enabling this communication.

(RPGs), and ITS based on Construction and Management Simulations, CMSs. These can be further divided into sub-genres – tycoon games being a specialization of CMS centering round the construction of infrastructures within a virtual organization. Genres may be mixed, e.g. resulting in CMS with role-playing elements.

A game genre is defined by certain design conventions, e.g. sets of typical game rules and the activities the player performs. Some typical activities in RPGs are the crafting of items or conversation with Non Playing Characters, NPCs. In an educational game – or game-based ITS, in our case – crafting would work well when teaching engineering skills, while conversations are useful in the humanities, e.g. interacting with characters in a game version of a classical drama. Typical activities in a CMS game include acquisition, production, sale, construction, and management. Naturally, these activities are particularly suited for teaching sub-domains of economics.

From a software design perspective, the game mechanics of a certain genre acting as a basis for adaptive educational games constitute a specialized instantiation of our framework. Game functionalities distribute among semantic components and modules. Game activities as a central form of established gameplay conventions draw on several of the basic functionalities. Software designers can therefore regard them as communication patterns – in fact, composed communication patterns, as they can build on the two atomic patterns introduced above.

Figure 2c illustrates a basic version of conversation as a communication pattern in our JaBInT-based adaptive educational RPG architecture. The player acts in a VirtualWorld where his or her Player Character, PC, can engage in conversation with NPC’s. The VirtualWorld module sends a PC’s question to the NPCManager. It fetches the NPC’s answer from the NPCRules via a request communication and forwards it to be presented in the VirtualWorld. This will repeat until the end of the conversation. The pattern already includes a step of reflection on the exchanged content – educational content in an educational RPG – by providing port connections to the CharacterSheet, where the conversation is recorded and possibly annotated.

We have not yet included connections for adaptation or intelligent tutoring that would determine questions available to an individual user or how a NPC reacts to a particular PC.

Figure 2d illustrates a basic version of the crafting RPG pattern. It differs from the conversation pattern in that it involves three modules within ProcessSteering and communication in-between. The ItemManager sends an instruction to EnvironmentManager notifying it that environment parameters have changed. This reflects a typical game scenario where a player modifies the environment using a tool, e.g. cutting a tree with an axe in the process of crafting a wooden boat.

Again, we have so far omitted an adaptation part where e.g. the character sheet would store which recipes the player has learned and which new recipe the tutor should present next.

Figure 2e shows the management pattern as one example of communication patterns in an educational CMS game architecture. The player composes a business plan in the only User Interface module CompanyWorld, which sends it to the EconomyManager for validation. This is done through communication with the EconomyRules in the Expert semantic component. The validated business plan is then returned to the user via the CompanyWorld, and the old business plan stored in the User component (Competition) is updated. This pattern does already allow for adaptation by an intelligent tutor. EconomyManager may consult the modeled Competence, before sending off the business plan to be validated. One application of this communication might be to check where to annotate the business plan in order to help an individual learner.

We have identified similar patterns for the other CMS activities. In a turn-based (i.e. discrete, not real-time) CMS, we also consider an overall game turn involving all user-initiated planning activities and their simulation along with scoring during that turn as a pattern. As a final note, all three presented game patterns exhibit the idea of User and Expert as the "model", UserInterface as the "view", and ProcessSteering as the "controller", where port communication mediates between these classical architectural layers.

6 Conclusions and Future Work

This article emphasizes the importance of communication in component-based software engineering, specifically in the context of Intelligent Tutoring Systems. While typical components in ITS have long been identified and have been translated into a framework architecture recently, this makes just for half of the design work. Designing communication flow between components or modules is tricky. By starting to collect and document design patterns, this article contributes to facilitating the work of software designers in the Technology Enhanced Learning field.

"Usually many steps are required to check whether a user entry is correct and to retrieve a feedback for the learner."
We are going to collect more communication patterns, structure this collection, and assemble all patterns into a catalogue. Figure 4 shows a first overview that includes most of the patterns discovered so far. It uses Unified Modeling Language, UML, syntax to document composed patterns (aggregation, diamond) as well as special patterns that extend (inheritance, triangle) more general patterns. The two atomic patterns (cf. Figure 2a) are displayed in the centre. All composed patterns are made up of these, sometimes only one type, sometimes of both. The bottom left of Figure 4 shows secondary window, which realizes a user interface where different modules present information in different display areas (the secondary windows), contained in one container area (the primary window). Communication between hierarchically organized windows can be seen as a specialization of communication between hierarchically organized modules. User interaction with a secondary module is handled by the pattern user entry secondary window, which can be further composed with user entry check feedback (Figure 2b) to result in the rather complex pattern user entry secondary window check feedback.

The top and the right-hand side of Figure 4 situate the special-purpose game patterns in the catalogue draft. Here, we can use abstract generalizations (pattern names in italics). These will never be implemented alone; instead they provide communication flows shared by their specializations. In the case of educational CMS, acquisition, production, sale, and management all use the same ports and connections, merely exchanging different message data. Each activity in an educational RPG (Figure 2 c-d) may result in a learning journal entry in the CharacterSheet module. The catalogue draft models this by a pattern journal entry that is a part – by composition – of any RPG activity.

Acknowledgment

We thank Alexander Neef for his work on communication patterns in component-based construction and management simulation games. The communication pattern project is part of the Research Training Group "Integrative Development of Modeling and Simulation Methods for Regenerative Systems" (diEM eSiRiS). The authors thank the DFG (Deutsche Forschungsgemeinschaft) for its financial support.

References