From Behavioral Description to A Pattern-Based Model for Intelligent Tutoring Systems

Javier Gonzalez-Sanchez, Maria Elena Chavez-Echeagaray,
Kurt VanLehn, Winslow Burleson
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University Tempe, Arizona, US
University Drive and Mill Avenue, Tempe, AZ 85287
+1 (480) 965-9253
{javiergs, helenchavez, kurt.vanlehn, winslow.burleson}@asu.edu

Abstract. Intelligent Tutoring Systems are capable of becoming an alternative to expert human tutors, able to provide direct customized instruction and feedback to students. Although Intelligent Tutoring Systems could differ widely in their attached knowledge bases and user interfaces (including interaction mechanisms), their behaviors are quite similar; thus, it must be possible to establish a common software model for them. A common software model is a step forward to move these systems from proof-of-concepts and academic research tools to widely available tools in schools and homes. The work reported here addresses: (1) the use of Design Patterns to create an object-oriented software model for Intelligent Tutoring Systems; (2) the application of the model into a two-year development project; (3) the qualities achieved and trade-offs made. Besides that, this paper describes our experience using patterns, and the impact in facts such as creating a common language among stakeholders, supporting an incremental development and adjustment to a highly shifting developing team.

Keywords: Design Patterns, Intelligent Tutoring Systems, Behavioral Description, Object Oriented, Model.

1 Introduction

The use of Intelligent Tutoring Systems (ITS) is becoming more common and there is a lot of work about their pedagogical and instructional design [4][16][2][21] but not about their technological implementation. This paper describes our approach to address the technological implementation of ITS within a context driven by three key elements:

a) **Incremental requirements.** This was a two-year project with incremental software requirements to implement the ITS in a components-oriented approach to support the integration of meta-tutoring and affective learning companions.

b) **Changing requirements**. It was required to test several and diverse research approaches, as part of the project implementation, creating for each approach a solid system able to be delivered in students' computers.

c) **Changing Team**. The programming team composed of undergraduate students shifted constantly (every 4 to 6 months).

In this context we made our choice to use Design Patterns to standardize a model for ITS functionality that drives the way in which software was developed. Design Patterns provided us with a common vocabulary, and help us to reduce system complexity by naming and defining abstractions for building reusable components from which more complex components were built [9].

We took as specification for ITS functionality the behavioral description given in [17], that claims that few pedagogical features have been invented, and that the different Intelligent Tutoring Systems, developed before now, offer different combinations of those features. The behavior of ITS described in [17] was produced from the analysis of different Intelligent Tutoring Systems currently available.

We mapped the functional description of ITS behavior described in [17] into a software model using the "Gang of Four" (GoF) Design Patterns [8], and we proposed this model as a software model for new Intelligent Tutoring Systems implementation. Using GoF Design Patterns we addressed the creation of the model and look forward to incorporate non-functional elements (i.e. software quality factors) particularly reusability, extensibility, and flexibility [11]. These qualities help us to address the contextual elements mentioned above: incremental requirements, changing requirements and changing team.

With this approach we are attempting to make our contribution to move ITS construction from software development as a one-of-a-kind endeavor to software development as a system of components that are widely used and highly adaptable [12]. The work described here is part of the development of an ITS, named Affective Meta Tutor (AMT) [1]. AMT project, funded by the National Science Foundation, is about including in an ITS, meta-tutoring strategies and affective learning companions technology. AMT project looks to improve ITS not only by adding new elements, but by taking advantage of previous experiences of Intelligent Tutoring Systems implementations from the ITS community. Most of our knowledge of those previous experiences is taken from the analysis and comparison made of existing Intelligent Tutoring Systems described in [17].

This paper is organized as follows: section 2 provides some terminology and background about ITS and patterns; section 3 explores ITS functional specification and the design process using patterns to model ITS software components; section 4 describes our experience using our pattern-based model into the AMT project and evaluates pros and cons; finally, section 5 concludes the paper and describes ongoing work.

2 Background

This section provides background about ITS structure and clarifies some related terminology used within this paper. It also provides background information about Design Patterns and the definition of the software qualities expected for the proposed model.

2.1 ITS Structure

ITS refers to a computer system that acts as a tutor showing an intelligent way to provide feedback and hints to support student achievement. ITS structure can be represented as a three-tier model, as shown in Figure 1, that decouples from the ITS Core, the Knowledge Base and User Interface.

a) **Knowledge Base (KB)** includes data structures and databases responsible for putting into the computer system the information instructed by the ITS. The process of putting data in KB is called "authoring". Authoring involves a human expert interacting with an authoring tool to provide this knowledge. Occasionally machine-learning algorithms has been used to create this expertise. Authoring and Knowledge representation are topics outside of this paper.

b) **User Interfaces (UI)** includes graphical interfaces (windows, buttons, text and so on) and interaction mechanisms (from simple keyboard events to more complex interfaces, such as motion capture, voice recognition, brain-computer interface and so on).

c) **Core** implements ITS behavior. While Knowledge Bases and User Interfaces are highly different from one ITS to others, the behavior of all of them is quite similar and the next components can be identified: (1) Task Selector provides a Task (problem or activity) the student must solve; (2) a Tool or Environment presents the information that the student must know to complete the activity; (3) Step Analyzer methodically examines and measures the performance of the student and provides that information to the Assessor and the Pedagogical module; (4) Pedagogical Module provides support (hints and feedback) to make the student successfully complete the task; (5) Assessor learns from the student (how many hints he needed, how skilled was in the topic, how much time he used to go from one step to another in order to solve the task, etc.) and then stores this information in what is called a Learner Model.

As examples of Intelligent Tutoring Systems we have Algebra Cognitive Tutor (an ITS for Algebra in High School) [2], Andes (a tutor for Physics in College) [18], AutoTutor (also an ITS for physics in College) [10], Sherlock (a simulator of avionic electronic equipment) [13], and SQL-Tutor (an ITS to teach SQL language) [15]. They were analyzed and compared in [17] where it is stated that their behaviors (and in consequence their Cores) are similar but they differ widely in their software implementation.

From a software engineering perspective, this shows a lack of the use of software engineering techniques and methodologies in the development of this kind of systems, because the same specifications are creating different products. Subsequently, it will be valuable to establish a model to go from the ITS behavior description to the system implementation. An optimal model should be capable of satisfying the requirements of these Intelligent Tutoring Systems and providing desired software qualities. The rest of this paper is related to modeling the ITS Core, the layer that implements the behavioral response of the ITS. Modeling Knowledge Bases and User Interfaces is out of the scope of this paper.

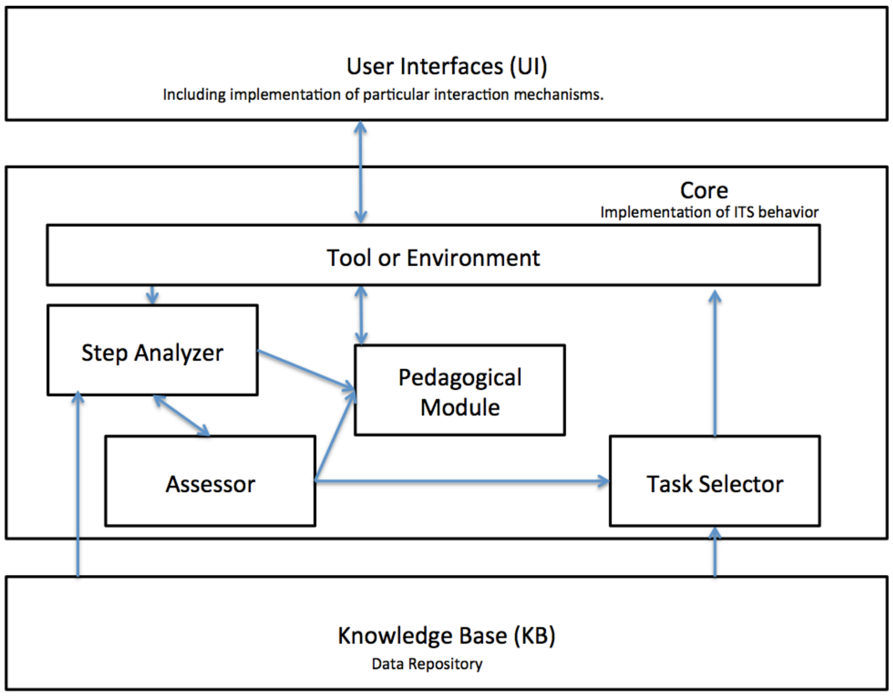


Fig. 1. ITS structure: User interface (and interaction mechanisms), Functionality (Core) and Data (Knowledge base) are decoupled.

2.2 ITS Terminology

For terms related with the ITS structure, mentioned before and used in the rest of this paper, the following list states their meaning:

a) **Task** refers to a multi-minute activity assigned to the student by the ITS. Tasks can be skipped or interchanged with other tasks.

b) **Step** is each of the actions taken to achieve a Task. Each Task consists of multiple Steps and each Step involves events with the user interface (either through a tool or an environment).

c) **Knowledge Components** are fragment of persistent, domain-specific information that should be used to accomplish a Task. Knowledge Components are contained in the Knowledge Base.

d) **Outer Loop** is the generic name given in [17] to the ITS process made by the Task Selector. Task Selector creates and chooses Tasks to be accomplished by the Student to become skilled in a particular Knowledge Component.

e) **Inner Loop** refers to the name given in [17] to the ITS process made by the Step Analyzer (which deals with the Steps of the chosen Task). This involves the Pedagogical Model, which provides Help (hints and feedback), and the Assessor that assesses the student performance and creates the Learner Model.

This terminology refers to a behavioral description, so the word "loop" must not be interpreted as a programming loop. Outer Loop and Inner Loop are names for two important processes accomplished by the ITS.

2.3 Why Patterns?

Software Design Patterns are used as a general reusable solution to a commonly occurring problem in software design, to show relationships and interactions between components and provide a skeleton for the implementation [8]. Even though, the concept of patterns has received little attention so far from researchers in the field of ITS, in [7] they mention that many Intelligent Tutoring Systems designers and developers use their own solutions when faced with design problems that are common to different systems, models, and paradigms; even when a closer look into that solutions and their comparison often shows that different solutions and the contexts in which they are applied have much in common.
In that context, our choice about using Design Patterns into this project was driven by our interest in:

a) **Communication**. Patterns provide us with the description of the topology of the system and the structural hierarchy of the subsystems and their interfaces and connections. Patterns are more abstract than just a technical model, but more technical than a conceptual model.

b) **Collaboration**. Patterns support the sharing of constructions between developers or either use other's constructions to enhance our own. No matter what is been built or what others built, it is always known which are going to be the relations (connections) among different constructions.

c) **Creativity**. Patterns help to create components, and components support the creation of families of products and/or several versions of the same product to prototype and test new options of functionality.

d) **Abstraction**. Using patterns it is possible to provide a "controlled" freedom to the programmers. They can develop functionality in their own creative way, but they follow and preserve the guidelines of a defined design.

These benefits of using patterns (communication, collaboration, creativity and abstraction) help us to overcome the challenging contextual elements of the project (incremental requirements, changing requirements and a changing team).

2.4 ITS Qualities

One additional reason to use Design Patterns is related to Quality. Software quality criteria are specified as non-functional requirements. Patterns let us take advantage of previous experiences to implement non-functional requirements and to avoid accidental complexity. Modeling ITS behavior is also about accomplish important non-functional considerations that drive its design. Non-functional requirements addressed in this paper are:

a) **Reusability** refers to the degree to which a software module or other work product can be used in more than one computer program or software system [11]. ITS components must be able to be used again with slight or no modification, for the implementation of other products or versions of the same project.

b) **Extensibility** is the degree to which a system or component can be easily modified to increase its storage or functional capacity [11]. ITS components in the model must be able to incorporate new functionalities or modify existing functionalities. By example assessment strategies, task-creation strategies, learning algorithms to mining learner model, etc.

c) **Flexibility or adaptability**. The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [11].

d) **Robustness** is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [11]. Students expect to get effective and efficient support from the ITS, as if it was a human tutor; interruptions in the teaching-learning process due to software failures are highly undesirable.

e) **Performance** refers to the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage [11]. The ITS must emulate real-time responses from a human tutor; delays must be avoided and latency reduced.

The use of patterns becomes the keystone to satisfy the first three qualities enumerated above. Satisfaction of both performance and robustness requirements are related to the implementation of the model and not with the model per se. However, in our experience communication, collaboration, creativity and abstraction impact performance and robustness.

3 Modeling the ITS Behavior

This section uses the ITS behavior described in [17] to create a conceptual model for ITS Core layer. The ITS behavior stated in [17] is summarized with a list of statements which identifies the involved components, responsibilities for each component, and relationships between components. In the list, components' names were marked in bold and relationships between components were explained. Complex components were split into simple ones, identifying specific responsibilities and assigning them to new components. The list of statements is as follows:

a) **Tool** is the component that recreates an environment for the **Student** to work. **Tool** handles the events fired by User Interfaces.

b) The ITS behaviors start in the **Outer Loop**.

c) During the **Outer Loop Task Selector** place a **Task** into the **Tool** in order to be solved by the **Student**.

d) **Task Selector** main responsibility is selecting the next **Task** that the student must solve. The selection is done in an "intelligent way"; four basic methods to do "selection" are described in [17].

e) **Task Selector** needs to have access to a source of **Task**s. We defined **Task Factory** as that source of **Task**s.

f) **Task Factory** creates **Task**s. Creating a **Task** means reading **Task**s stored in a repository (read previously human-authored **Task**s) or creating **Task**s in real-time.

g) **Task Selector** relies on **Learner Model** to choose a **Task**.

h) **Learner Model** is maintained by the **Assessor** component**.**

i) **Inner Loop** is nested inside the **Outer Loop**. The **Inner Loop** works with the **Step**s that conforms the **Task**. In the **Inner Loop** participate the **Step Analyzer**, the **Assessor** and/or the **Pedagogical Model**.

j) **Step Analyzer** assesses the **Student** performance while collecting and processing data about the student's learning process.

k) **Assessor** looks at the information generated by **Step Analyzer** and store it in the **Learner Model**. The accuracy of the diagnostic algorithms of this component is a key factor for the adaptation process.

l) **Pedagogical Module** provides **Help** using different strategies such as providing immediate or delayed help, or providing requested or unsolicited help.

m) **Steps** include **Assessment** and **Help**.

n) **Help** could be **Hint**s before completing the **Step,** or **Feedback** after completing the **Step**.

o) **Task** is a set of **Step**s.

p) **Task** is related to a set of **Knowledge Components**.

q) **Learner Model** is a set composed of **Task**s, measures of the time spent to complete the **Task** and the status of the **Task**. For each **Step** in the **Task** a counter of the **Hint**s requested and **Feedback** (errors made) is kept, and for each **Knowledge Component** a mastering measure is also kept for each **Student**.

r) **Knowledge Components** are the information and skills being taught.

s) **Knowledge Base** is the set of **Knowledge Components** in the ITS.

Figure 2 extends ITS structure showed in Figure 1 in order to identify components (functional and data objects) and their relationships. UML notation is used, inside ITS Core block, to create a first attempt of the object-oriented model. Figure 2 shows:

a) Each component is represented as a box: white boxes are functional components and gray boxes represent data components.

b) Association relationships are shown using arrows; the arrows go from the component that requests a functionality to the component that provides that functionality. Some examples are: Tool sends information about events in the User Interface to Step Analyzer; Task Selector uses Tool to present a new Task; Task selector uses Task Factory to obtain Tasks.

c) Dependency relationships are represented using arrows with dashed lines; in the model we are showing the dependency between functional components and the data component they require to access them.

d) In data components (gray boxes) inheritance relationships (arrows with a triangular shape in the arrow point) show a specialization hierarchy; in our case this relationship shows that Help could be either a Hint or Feedback.

e) Finally, composition relationships are shown with arrows starting with a diamond shape. Specifically, they are used to represent Task as a composition of Steps, and Knowledge Base as a composition of Knowledge Components.

The next section explains how this conceptual model became an object-oriented model, using a pattern-based approach.

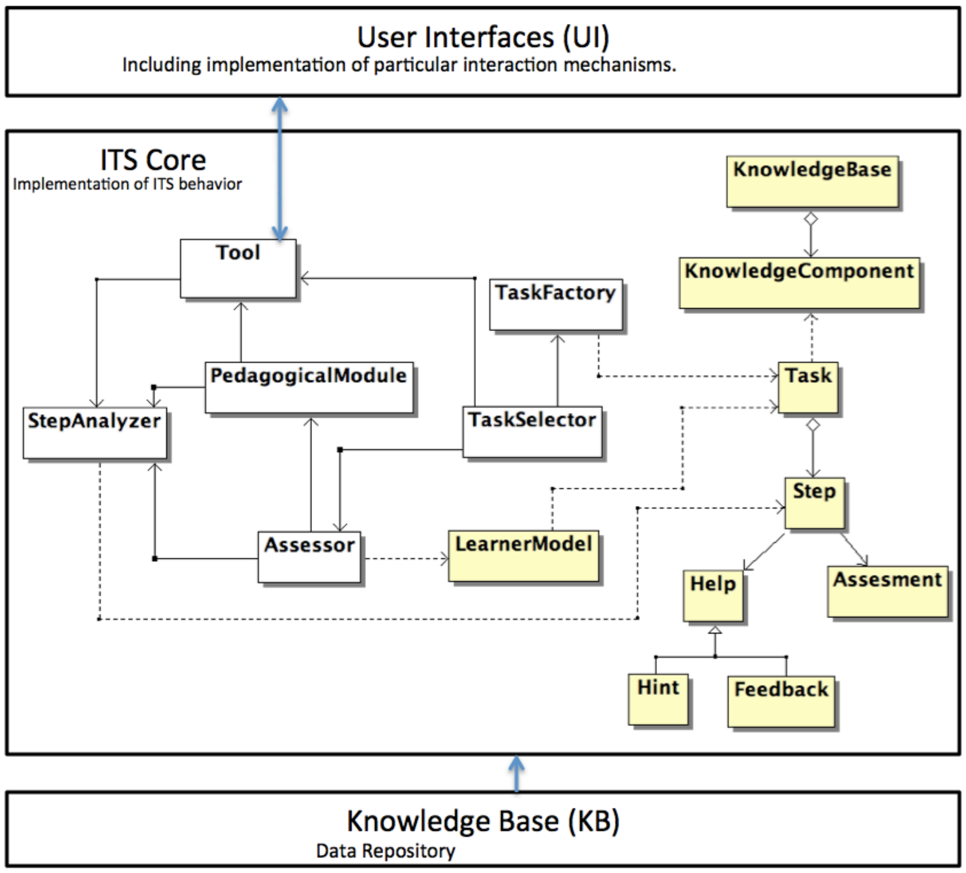


Fig. 2. ITS Conceptual model. White boxes show functional components and gray boxes show data components. Relationships of association, dependency, composition and inheritance are showed using UML notation.

3.1 Pattern-Based Modeling

The previous model showed in Figure 2 and extracted from the behavior described in section 3 represents a conceptual description of who is doing what, and corresponds to an abstraction of the expected functionality of each internal component in ITS Core. Even though the conceptual model can be a starting point to implement ITS functionality, it is still too abstract to be a software design and therefore there are a lot of diverse options to implement it. The next step in our process was to evolve this model by defining more specific relationships between components, using Design Patterns; that provided us with a template for the software design and thus for the implementation.

Finding the appropriate pattern to be applied for each component and relationship was a process based on experience and literature research [6][8]. There is not a rule about how to choose a pattern, it is required to know the existent patterns (the problems they solve) and then use them to describe in an effective way what is happening in the system. Our approach consists of using the pattern that most closely matches the semantic description of the requirement or group of requirements. From the "GoF" Design Patterns documented in [6] and [8] we took the keywords: observer, abstract factory, builder, singleton, chain of responsibilities, strategy, communicator, facade, composite and singleton; each pattern are fairly close to implement what their name means and what our components are supposed to do. For example, Task Factory naturally means to be an **ABSTRACT FACTORY** of Tasks. Table 1 shows the relationship between components previously defined matched to a pattern name with a description of the meaning of the relationship

Table 1. Relationships between ITS components and Design Patterns

Components	Pattern	Description
Tool	**FACADE**	Tool is a high-level interface for a set of subsystems.
TaskSelector	**STRATEGY**	TaskSelector component is implemented using **STRATEGY** pattern to lead with the fact that selecting the next Task for the Student is done with different algorithms (methodologies), described in [11].
TaskSelector and Assessor	**OBSERVER**	The relationship between TaskSelector and Assessor can be described by **OBSERVER** pattern. TaskSelector needs information about changes in the Learner Model (performance of the student) maintained by Assessor component, in order to adjust the level of the next Task.
TaskFactory	**ABSTRACT FACTORY**	TaskFactory creates Task objects. The relationship between TaskFactory and Task corresponds to the relationship between a factory and a product in **ABSTRACT FACTORY** pattern.
TaskFactory	**STRATEGY**	TaskFactory implements **STRATEGY** to create Tasks, due to the fact that ITS could implement either particular algorithms to create Tasks in real-time, or create Tasks recovering them from a data repository.

StepAnalyzer	**CHAIN OF RESPONSIBILITIES**	**CHAIN OF RESPONSIBILITIES** is a design pattern that avoids coupling the sender of a request to its receiver, by giving more than one object a chance to handle the request. StepAnalyzer chains the receiving objects and passes the request along the chain until one handles it. Since Step Analyzer works with Steps, and Steps are close related with user events, modeling Step Analyzer as a chain gives us the opportunity to add and remove behavior associated with specific events quickly.
Assessor	**STRATEGY**	Assessor implements **STRATEGY** to maintain the Learner Model. Diverse strategies could be tried to store and recover the Learner Model information.
Pedagogical Module	**STRATEGY**	Pedagogical Module implements **STRATEGY** to provide support to the student in solving the current Step. Options to provide Help go from pressing a button asking for a Hint, to the implementation of intelligent algorithms that provide support to maintain the student in the "zone of proximal development" [20] where tasks are neither boringly easy nor frustratingly difficult, but instead afford maximal learning and motivating challenges.
Step	**COMPOSITE**	Step uses **COMPOSITE** pattern to compose Steps into tree structures to represent part-whole hierarchies. **COMPOSITE** lets us treat individual Steps and hierarchies of Steps (and sub-Steps) uniformly.
Knowledge Base	**SINGLETON**	Making KnowledgeBase a **SINGLETON** ensures only one instance of it and provides a global access point to it.
Assessor and StepAnalyzer	**OBSERVER**	The relationship between Assessor and StepAnalyzer can be described by **OBSERVER** pattern. Assessor needs information about student performance in each Step. That information is obtained from StepAnalyzer.

Pedagogical Module and StepAnalyzer	**OBSERVER**	The relationship between Pedagogical Module and Step Analyzer can be described by **OBSERVER** pattern. Pedagogical Module needs information about student performance in each Step. That information is obtained from StepAnalyzer.

3.2 Putting All Patterns Together

With the relationships expressed as pattern equivalences, as listed in the previous section, creating a software design is fairly straightforward. Each pattern has a unique equivalence in UML (as a class diagram). Converting the UML class diagram into code files could even be an automatic process done by a development tool. Then, our programming team would be able to focus on the detailed implementation of the desired functionality, filling in specific places inside specific files, methods and attributes [5].

The UML class diagram for the ITS Core layer is shown in Figure 3. It is important to note the following relationships in the diagram:

1. TaskSelector, TaskFactory, Assessor, and PedagogicalModule are implementing the **STRATEGY** pattern that permit us to define algorithms, encapsulate them, and make them interchangeable. **STRATEGY** pattern lets the algorithm vary independent of the classes that use it. Implementing a new way to select a Task, create a Task, manage the Learner Model or provide Help to the student can be done for one developer who needs no knowledge about the project at all; the developer just needs to follow the pattern to: (1) create a new class that implements the corresponding interface; (2) implement at least the algorithmInterface() method; and (3) create as many additional methods and/or attributes as needed.

   At AMT project: for TaskSelector only a SequencialTaskSelection strategy was defined; for TaskFactory, a TaskFromRepository strategy was implemented. No strategy was implemented for Assessor (this module is still an ongoing part of the project); for PedagogicalModule a ConditionalPedagogicalStrategy strategy was implemented in which conditionally the presence of certain events or actions from the Student launches pre-established responses of PedagogicalModule.

   In addition to this, a first version of a MetaTutor System was implemented and linked with a MetaTutorPedagogicalStrategy class. The MetaTutorPedagogicalStrategy class acts as an **ADVOCATE** sending and receiving information from and to the external system (MetaTutor System). Those implementations are not shown in the diagram because of space limitations.

2. TaskSelector obtains information from Assessor, which as well as PedagogicalModule obtains information from StepAnalyzer. The concept of “observing” describes the relationship and clearly identifies how the structure of

communication must be implemented (methods and attributes). It is easy to notice which component needs information from which other component.

3. The implementation of Step as a `COMPOSITION` is highly useful, it provides the capability of managing Steps as one or as a hierarchy of several hierarchized Steps.

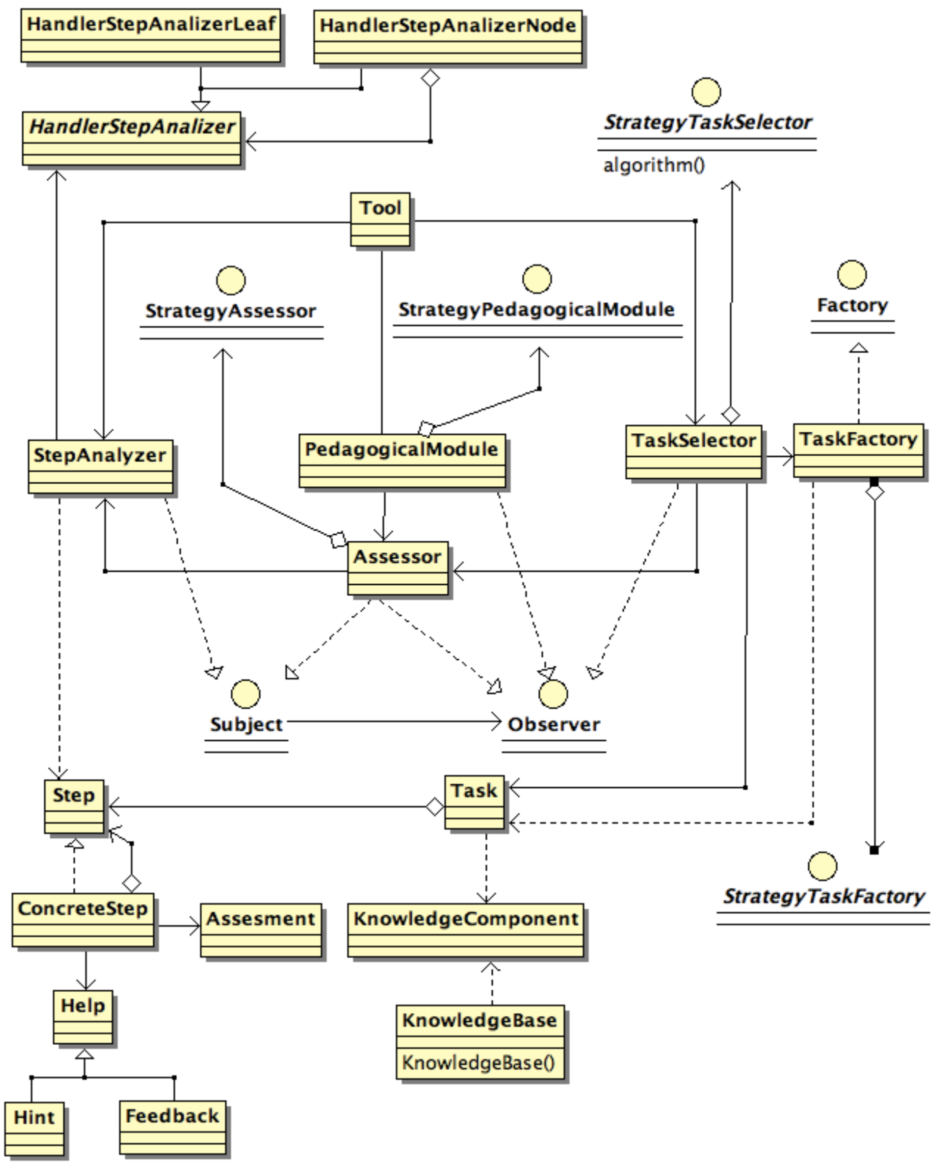


Fig. 3. UML class diagram showing the pattern-based model for the ITS Core layer.

For AMT project Tool component is an environment in which Student is able to learn about systems dynamic modeling, using graphical representation. Each model is a directed graph formed by nodes and edges. The edges indicate flow of numeric information between nodes and the nodes represent variables. A node encapsulates a variable's value as an algebraic combination of the numbers coming into or going out of it via edges. Students read text describing the system, and then define nodes and edges, enter value or equations in each node, run the model and compare its predictions to given facts. If any of the model's predictions are false (represented with red colors as feedback), students must debug the model. Students also can ask for feedback by checking their model at each step before running the model. [19].

Figure 4 shows the current implementation of AMT Tool component. Tool component consists on a Canvas in which a Graph is drawn. A Graph is composed by Nodes (Vertex) and Links (Edges) that connect the Nodes. Each Vertex maintains a register of all vertexes going out and in. Each Edge maintains data of the Vertex in which it starts and ends.

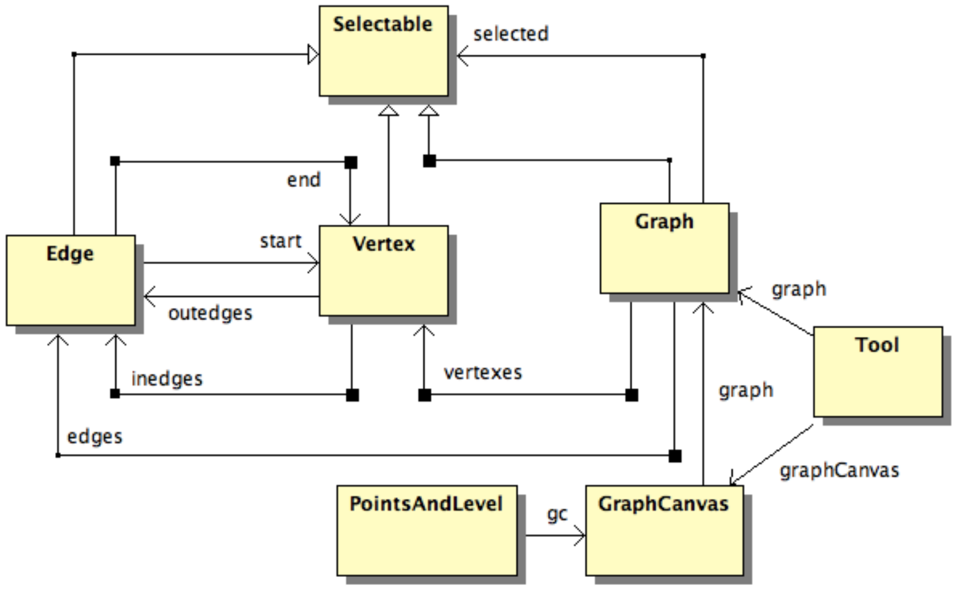
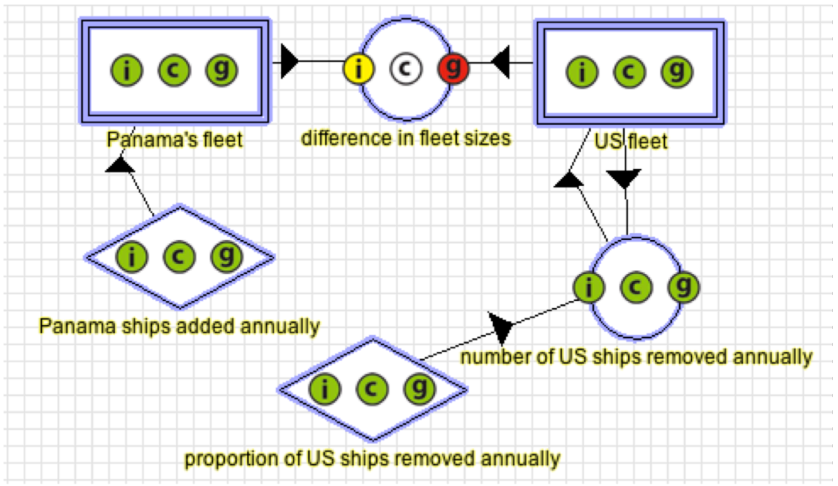


Fig. 4. UML class diagram showing the Tool implemented in AMT project, encapsulated in the model as a Tool by FACADE pattern.

Vertex, Edges and Graph can be selected from the Canvas and manipulated (drag and drop, deleted, and so on). The Tool in execution is showed in Figure 5. The figure shows the solution for a problem about Merchant marine that states:

After World War II, the United States had the largest merchant marine of any nation. The merchant marine are the ships that transport goods and people over the oceans, not counting navy vessels. Unfortunately, the US merchant marine has been getting smaller and smaller each year, while the merchant marine of other countries *has grown. Just for illustration, suppose that in 1950 the US merchant marine was 5,000 ships of 10,000 tons or larger, where Panama's merchant marine was only 1,000 ships of 10,000 tons or larger. Suppose that the US Merchant marine shrank by 5% each year, mostly because ships were sold to other countries. Suppose the Panama's merchant marine grew by 100 ships a year, mostly because it was cheap to own a ship registered in Panama. Graph the difference in size between the two fleets over 50 years. That is, the difference starts out with the US having 4000 more ships than the Panama in 1950. What happens on the way to 2000?*

The complete model built by a student that shows the difference in size between the two fleets (Panama's and US's) described in the text above. This difference is calculated as a function of the number of ships on each fleet, which also depend on the number or ships added or removed annually. It is possible to observe that the student obtained all the names (labels) for the nodes from the tool (observe the yellow color), however he correctly defined the type for each node, almost all their inputs (i), as well as almost all the equations or values (c) for each node (see green indicators). Observe that the student did not check if the equation defined for the node that represents the difference between the fleets was correct or not, this can be told due to the fact the (c) indicator is white. When the student checked (run) his model, he obtained the feedback for the graphics for each node (g), as it is shown the student have a problem in the node that represents the difference on the fleets (red indicator).


i
c
g
Panama's fleet
difference in fleet sizes
US fleet
Panama ships added annually
number of US ships removed annually
proportion of US ships removed annually

(b)

Fig. 5. Look and feel of the AMT Tool. GraphCanvas is a UI component where the model is displayed and could be manipulated. Model is composed by Vertex (geometrical shapes) and Edges (arrows). The colors represent the Feedback from the student about the status of each Step.

4 Experience Report and Evaluation

Including structure and functionality, the current system, developed in Java, is formed by: 10 packages; 62 classes; 746 methods; 738 attributes; 22,434 lines of code; 1,150 revisions maintained in a revision control system (SVN) created between July 2009 and July 2011 for a shifting team of nine programmers (maintaining a team of two programmers at a time, with an average of six months of permanency) and two resident software engineers; 8 versions released to clients; and 140 users working with the system, who have been high school students and undergraduate students participating in four summer camp courses and two university courses at Arizona State University.

Our experience using Design Patterns to create an ITS model and implement it to create AMT software project can be summarized as follows. Stakeholders in general mentioned as points in favor:

a) **Incremental development fully supported**. Our goal to build the AMT project in an incremental way, during two years has allowed us to: (1) provide in a window time of two or three weeks a new product or a new version of the product; (2) achieve time reduction to deployment with more programmers, in specific moments of the project, when time is related with the implementation of new functionality.

b) **Changing programming team almost twice a year**. (1) Programmers, even without knowledge of patterns, are able to focus their attention in the requirements assigned to them; we used Subversion [3] to maintain a common repository of the project, each programmer works in completing a specific module or set of components (defined as a pattern section); (2) relationships between components are almost fully defined by patterns connections, so the work done by each programmer is delimited and merging the work from different programmers is a straightforward process.

c) **Vocabulary**. The use of pattern names such as **`FACTORY`** and **`STRATEGY`** has been adopted as an abstract way to refer functionalities between stakeholders. The names hide complexity from non-developers. Non-developers assume an easy thing must be done, and programmers have a better idea about the boundaries of changes, bugs and new requirements.

However some dispute did emerge regarding:

a) **Size**. It is arguable, but some people point to the increment of the size of the code while using patterns. Yes! It could be true, using patterns generates more code, but it is not only due to patterns (interfaces and abstract classes declarations), but also because we decide to maintain the cyclomatic complexity (McCabe number) [14] for every method under 10, which means an applied "divide and conquer" strategy, and that generates more methods in the system.

b) **Time delays**. Unlikely others approaches our first step, even before showing a prototype to the research group, was focused on the architecture definition (patterns). Thus, software prototypes delayed its appearance into the scene; but once the first prototype was presented, new prototypes emerged quicker that in previous projects.

4.1 Impacts of Design Patterns

It is important to specifically highlight how the use of Design Patterns impacted the project:

a) **Communication.** Since diverse stakeholders such as researchers in education technology, computer scientist, developers and instructional designers were involved, Design Patterns helped us to agree in the structure of the system and communicate it to the programmers for each individual component in the project.

b) **Collaboration.** Sharing constructions between developers was a key element to counterbalance the effect of a constant developers shifting.

c) **Creativity.** Creativity was encouraged, enhanced and achieved by allowing the creation of several versions of the project to prototype and test new options of functionality and in consequence, in our project, new pedagogical approaches.

d) **Abstraction**. Providing a "controlled" freedom to the programmers using patterns as the guidelines of a defined design was highly relevant to handle changing and incremental requirements.

5 Conclusions and Ongoing Work

Many authors claim that their ITS follow and accomplish a software architecture because they can identify components and relationships among those components inside their systems. However, this does not mean that standard and good practices, such as Design Patterns, have been followed.

We take advantage of the growing experience in the field of software Design Patterns to both design and implement an ITS model in a pattern-based approach. Applying Design Patterns was useful to create a high-quality software solution that is easy to maintain and extend.

Designing with quality attributes as drivers, has resulted in a design that has proven to be more modifiable, reusable and reliable. Using Design Patterns impacts our communication, collaboration, and creativity. Design Patterns facilitate the adjustment of a highly shifting programming team and thus the development of the system, where the creation of new versions or variants of the software was relatively easy in terms of time and effort. Adding Design Pattern in the development of ITS allowed us to create a common vocabulary among stakeholders making the process more accurate and effective design-wise.

We applied our model to build several variants of AMT system in two years of work, with a high rate of changes in requirements for the product and a changing programming team. In other words, we have been able to create a family of AMTs around the same design.

Future research will focus on two additions: (1) the first one will be the inclusion of a model for companions to provide support for the student, these are learning companions, affective companions and teachable agents; (2) the second one will be the inclusion of Meta-Tutoring components.

Acknowledgments

We are grateful to Hironori Washizaki for his support during the writing process of this paper.
This research was funded by the National Science Foundation, including the following grants: (1) IIS/HCC Affective Learning Companions: Modeling and supporting emotion during learning (#0705883); (2) Deeper Modeling via Affective Meta-tutoring (DRL-0910221) and (3) Pittsburgh Science of Learning Center (SBE-0836012).

References

[1] Affective Meta Tutor – Arizona State University. http://amt.asu.edu.

[2] Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. 1995. Cognitive Tutors: Lessons Learned. *Journal of the Learning Sciences*, 4(2), 167-207.

[3] Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. 2004 Version control with subversion. O'Reilly Media, Inc.

[4] Baker, R. S. J. D., de Carvalho, A., Raspat, J., Aleven, V., Corbett, A. T., and Koedinger, K. R. 2009. Educational software features that encourages and discourage "gaming the system". *Proceedings of the International Conference on Artificial Intelligence in Education*. IOS Press.

[5] Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J., and Houston, K. 2007. *Object-Oriented Analysis and Design with Applications*, Third Edition. Addison-Wesley Professional.

[6] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *A system of patterns: Pattern-oriented software architecture*. Wiley.

[7] Devedzic, V. and Harrer, A. 2005. Software Patterns in ITS Architectures. *International Journal of Artificial Intelligence in Education*, 15, 2 (April 2005), 63-94.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 2002. Design Patterns: abstraction and reuse of object-oriented design. *In Software pioneers*. Manfred Broy and Ernst Denert (Eds.). Springer-Verlag New York, Inc., New York, NY, USA 701-717.

[10] Graesser, A. C., Lu, S., Jackson, G. T., Mitchell, H. H., Ventura, M., Olney, A., et al. 2004. AutoTutor: A tutor with dialogue in natural language. *Behavioral Research Methods, Instruments and Computers*, 36, 180-193.

[11] IEEE. 1999. *Standard Glossary of Software Engineering Terminology*. 610.12-1990, Vol.1. IEEE Press.

[12] Jacobson, I.: 1997. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Professional

[13] Katz, S., Connelly, J., & Allbritton, D. (2003). Going beyond the problem given: How human tutors use post- solution discussions to support transfer. *International Journal of Artificial Intelligence in Education*, 13,79-116.

[14] McCabe, T. 1976. *A complexity measure*. IEEE Trans. Software Engineering, 5, 45–50.

[15] Mitrovic, A. 2003. An intelligent SQL tutor on the web. *International Journal of Artificial Intelligence in Education*, 13(2-4), 197-243.

[16] Nelson, B. C. 2007. Exploring the use of individualized, reflective guidance in an educational multi-user virtual environment. *In Journal of Science Education and Technology*, 16(1), 83-97.

[17] VanLehn, K. 2006. The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education*. Volume 16, Issue 3, Pages 227-265. IOS Press.

[18] VanLehn, K., Lynch, C., Schultz, K., Shapiro, J. A., Shelby, R. H., Taylor, L., et al. 2005. The Andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence in Education*, 15(3), 147-204.

[19] Van Lehn, K. et al. 2011. The Affective Meta-Tutoring Project: How to motivate students to use effective meta-cognitive strategies. T. Hirashima et al. (Eds.) *Proceedings of the 19th International Conference on Computers in Education*. Chiang Mai, Thailand: Asia-Pacific Society for Computers in Education.

[20] Vygotsky, L. S. 1978. *Mind in Society: The Development of Higher Psychological Processes*. Cambridge, MA: Harvard University Press.

[21] Wilson, B. G. 1997. *Reflections on constructivism and instructional design.* Instructional development paradigms. C. R. Dills and A. A. Romiszowski (Eds.), Englewood Cliffs NJ: Educational Technology Publications. 63-80.