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

JAVIER GONZALEZ-SANCHEZ, Arizona State University
MARIA ELENA CHAVEZ-ECHEAGARAY, Arizona State University
KURT VANLEHN, Arizona State University
WINSLOW BURLESON, Arizona State University

Intelligent Tutoring Systems are software applications capable of complementing and enhancing the learning process by providing direct customized instruction and feedback to students in various disciplines. Although Intelligent Tutoring Systems could differ widely in their attached knowledge bases and user interfaces (including interaction mechanisms), their behaviors are quite similar. Therefore, 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) our experience using this model in a three-year development project and its impact on facets such as creating a common language among stakeholders, supporting an incremental development, and adjustment to a highly shifting development team; and (3) the qualities achieved and trade-offs made.

Categories and Subject Descriptors: **D.2.10** [**Software Engineering**]: Design; **D.2.11** [**Software Engineering**]: Software Architecture; **D.2.13** [**Software Engineering**]: Reusable Software.

General Terms: Design.

Additional Key Words and Phrases: Design patterns, component model, intelligent tutoring systems, behavioral description.

ACM Reference Format:

Gonzalez-Sanchez, J., Chavez-Echeagaray, M.E., VanLehn, K. and Burleson, W. 2011. From Behavioral Description to A Pattern-Based Model for Intelligent Tutoring Systems. 18th Conference on Pattern Languages of Programs (PLoP), Portland, Oregon, USA (October 2011), 12 pages.

1. INTRODUCTION

Intelligent Tutoring Systems (ITS) complements and enhances the learning process by offering support for students and complementing the teacher's labor. There are many discussions about ITS pedagogical and instructional design, such as (Anderson et al. 1995, Baker et al. 2009, and Nelson 2007), but not about their technical implementation.

The work described here is part of the technical implementation of an ITS, named Affective Meta Tutor (AMT). The AMT project aims to use an affective learning companion to motivate students to permanently adopt effective meta-cognitive strategies (AMT 2012). The AMT project looks to improve ITS not only by adding those new elements (meta-tutoring strategies and affective learning companions), but also by taking advantage of previous experiences with ITS implementations from the ITS community to provide a software framework for designing ITS based on a behavioral description; modeling meta-tutoring capabilities and learning companions modules is not part of this paper and is left as future work.

An analysis and comparison of existing ITS was made in (VanLehn 2006). The analysis included a diverse set of ITS such as an ITS for Algebra in High School (Anderson et al. 1995); a tutor for physics in College

We are grateful to Hironori Washizaki for his support during the writing process of this paper. This work is supported by the National Science Foundation, including the following grants: (1) IIS/HCC Affective Learning Companions: Modeling and supporting emotion during learning (#0705883); and (2) Deeper Modeling via Affective Meta-tutoring (DRL- 0910221).
Author's address: Javier Gonzalez-Sanchez, University Drive and Mill Avenue, Tempe AZ 85287; email: javiergs@asu.edu; Author's address: Maria-Elena Chavez-Echeagaray, University Drive and Mill Avenue, Tempe AZ 85287; email: helenchavez@asu.edu; Author's address: Kurt VanLehn, University Drive and Mill Avenue, Tempe AZ 85287; email: kurthvanlen@asu.edu; Author's address: Winslow Burleson, University Drive and Mill Avenue, Tempe AZ 85287; email: winslow.burleson@asu.edu

(VanLehn et al. 2005); a tutor for qualitative reasoning in natural language (Graesser et al. 2004); a simulated-based tutor for repairing avionic electronic equipment (Katz et al. 1998); and an ITS to teach SQL language (Mitrovic 2003). The analysis and comparison concluded that only a few pedagogical features have been invented and that existing ITS offered different combinations of those features. It also claimed that ITS behaviors are similar but the ITS differ widely in their software implementation.

From a software engineering perspective, this variety in software implementation 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 that moves from the ITS behavior description to the system implementation. An optimal model should be capable of satisfying the requirements of ITS and providing desired software qualities. This paper describes our approach to address the design of this model within a context driven by three key elements:

- **Incremental requirements**. AMT required incremental evolution along three-years; developing an ITS the first year, adding meta-tutoring support the second year, and including affective learning companions the third year. Research results and user experience reports drove new requirements.
- **Changing requirements**. Several and diverse research approaches were tested as part of the project implementation. For each approach a solid system was released and tested with students. Research findings were translated into changes in the system.
- **Shifting development team**. The development team, composed of undergraduate students, shifted constantly - every 4 to 6 months.

In this context, we chose to incorporate design patterns to standardize an object-oriented model for ITS functionality that drives the way in which software is developed. We mapped the functional description of ITS behavior given in (VanLehn 2006) into a software model using some of the "Gang of Four" (GoF) design patterns (Gamma et al. 1995); GoF design patterns were chosen because they are classic patterns often considered the foundation for all other patterns. Using GoF design patterns we addressed the creation of the model and sought to incorporate on it non-functional requirements (i.e. software quality factors) particularly reusability, extensibility, and adaptability (IEEE 1999). These qualities help us to address the contextual elements mentioned above: incremental requirements, changing requirements, and a shifting development team. With this approach we seek to contribute moving 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 (Jacobson 1997).

This paper is organized as follows: Section 2 provides some terminology and background about ITS, patterns, and software qualities; Section 3 explores ITS functional specification and the design process using patterns to model ITS software components; Section 4 describes our experience using design patterns into the AMT project and evaluates pros and cons; finally, Section 5 concludes the paper and describes ongoing work. We expect developers in ITS and education technology communities to find this paper useful as a reference and as an example of the use and advantages of design patterns for developing software systems; for software design community, this is an experience report of a research group using design patterns to improve its software process.

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 while solving tasks. A task refers to a multi-minute activity assigned to the student by the ITS. Tasks can be skipped or interchanged with other tasks. Each of the actions taken to achieve

a task is called a step. Each task consists of multiple steps and each step involves events with a User Interface (either through a tool or an environment). Each task requires the use of one or more knowledge components to be accomplished. Knowledge components are fragments of persistent domain-specific information that the student is expected to learn. Knowledge components are contained in a knowledge base.

ITS structure is a three-layer model, as shown in Figure 1, that decouples the Knowledge Base and the User Interfaces from the Core of the ITS. The description of each layer is as follow:

- **Knowledge Base (KB)** includes data structures and databases for storing and organizing the information instructed by the ITS. The process of putting data into the KB is called authoring. Authoring involves a human expert interacting with an authoring tool to provide this data. Occasionally, machine-learning algorithms have been used to create this expertise. Authoring and KB representation are topics outside of this paper.
- **User Interfaces (UI)** include graphical interfaces (windows, buttons, text, and so on) and interaction mechanisms (from single keyboard events to more complex interfaces such as motion capture, voice recognition, brain-computer interface, and so on).
- **Core** implements the ITS behavior. While Knowledge Base and User Interfaces are highly different from one ITS to another, the behavior of all of them is quite similar. The Core is composed of: (1) Task Selector, which provides a Task (problem or activity) that the student must solve; (2) Tool or Environment, which models and presents the information that the student must know to complete the Task; (3) Step Analyzer, which methodically examines and measures the performance of the student and provides that information to the Assessor and the Pedagogical Module; (4) Pedagogical Module, which provides support (hints and feedback) to make the student successfully complete the Task; support is related with the performance of the student in the current Step and the information from the student's Learner Model; and (5) Assessor, which learns from the student (how many hints he needed, how skilled he 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 known as a Learner Model. The connections between the Core's components are described in Section 3.

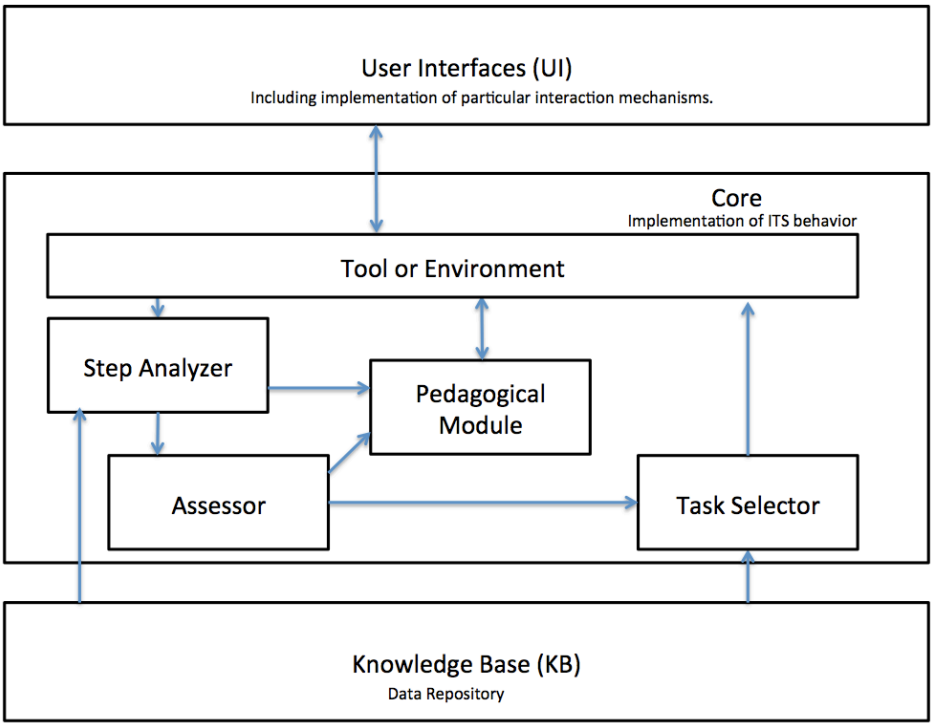

Fig. 1. ITS layered structure: User Interface (and interaction mechanisms), Functionality (Core), and Data (Knowledge Base) are decoupled.

The rest of this paper is devoted to modeling the Core, the layer that implements the behavioral response of the ITS. Modeling Knowledge Base, User Interfaces, and the connection between them and the Core is out of the scope of this paper. However, the connection between User Interfaces and Core can be easily figured out as an MVC pattern application (Buschmann et al. 1996), where Core acts as the Model part. The connection between Core and Knowledge Base can be realized using diverse data access approaches.

2.2 Why Design 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 (Gamma et al. 1995). Even though the concept of patterns has received relatively little attention in the field of ITS, Devedzic and Harrer (2005) mention that many ITS 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 in this project was driven by our interest in:

- **Communication**. Since patterns' names closely match their objective and the problem they solve, we used them as a common vocabulary among diverse stakeholders aiming to improve the communication process. Patterns provide us with a standard vocabulary to describe the topology of the system, the structural hierarchy of the subsystems, and their interfaces and connections.
- **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.
- **Productivity**. 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.
- **Abstraction**. Patterns are more abstract than just a technical model, but more technical than a conceptual model. Patterns make it possible to provide a "controlled" freedom to the programmers because 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, productivity, and abstraction) help us to overcome the challenging contextual elements of the project (incremental requirements, changing requirements, and a shifting development team).

2.3 ITS Qualities

ITS are pieces of software, hence they are expected to meet some software quality criteria. Therefore, modeling ITS behavior is also about accomplishing quality considerations that drive their design. Software quality criteria are specified as non-functional requirements. Accomplishing non-functional requirements is one additional reason to use design patterns. Design patterns let us take advantage of previous experiences to implement non-functional requirements and to avoid**,** when properly used, accidental complexity. The non-functional requirements addressed in the project were:

- **Reusability**. 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 (IEEE 1999). 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.
- **Extensibility**. Extensibility is the degree to which a system or component can be easily modified to increase its storage or functional capacity (IEEE 1999). ITS components in the model must be able to incorporate new functionalities or modify existing functionalities (e.g., assessment strategies, task-creation strategies, learning algorithms to mining learner model, etc.).

- **Adaptability**. Adaptability refers to the ease with which a system or component can be modified for using it in applications or environments other than those for which it was specifically designed (IEEE 1999).
- **Robustness**. Robustness is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions (IEEE 1999). 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.
- **Performance**. 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 (IEEE 1999). 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 the last two requirements (robustness and performance) is related to the implementation of the model and not with the model per se. However, in our experience communication, collaboration, productivity, and abstraction impact performance and robustness.

2.4 Unified Modeling Language

Unified Modelling Language (UML) notation is used as modelling language, specifically UML class diagrams. To facilitate the understanding of the diagrams offered in this paper (see Figures 2, 3, and 4) this section provides a brief description of the elements (boxes and arrows) within a UML diagram.

- Boxes represent components or classes.
- Arrows represent association relationships; the arrows go from the component that requests functionality to the component that provides that functionality.
- Arrows with dashed lines represent dependency relationships; in the model we are showing the dependency between functional components and the data component they require to access them.
- Arrows with a triangular shape in the arrowhead represent inheritance relationships.
- Arrows starting with a diamond shape represent composition relationships; they are used to represent that a component is formed by a conjunction of other components.

3. MODELING THE ITS BEHAVIOR

This section uses the ITS behavior described in (VanLehn 2006) to create a conceptual model for the Core layer. The ITS behavior, stated in (VanLehn 2006), is summarized in a list of statements that 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 are explained. Complex components were split into simple ones, identifying specific responsibilities and assigning them to new components. The list of statements is as follows:

- Students interact with the ITS through the **User Interface**. The events triggered by the **User Interface** are handled by the **Tool** component**.** The **Tool** is the component responsible for creating and managing the environment in which the student works.
- The ITS provides students with tasks to be solved and helps them in the process. A **Task** is a set of steps. For simplicity, this report assumes that the set of steps is static and pre-enumerated. In principle, a **Task** could be a step generator, which means steps could be dynamically generated. The ITS literature refers to tutors using a static, pre-enumerated set of steps as "example tracing tutors" whereas those using steps generators are called "model tracing tutors" (VanLehn, 2006). Each **Step** is related to events in the **User Interface**. Steps include **Assessment** and **Help**. **Help** could be **Hints** before completing the **Step** or **Feedback** after completing the **Step**. Each **Task** is related to a set of **Knowledge Components** that are the information and skills that a student needs to apply in order to solve the **Task** successfully. The **Knowledge Base** is a set of **Tasks** and **Knowledge Components** and the mapping between them.
- The ITS behavior starts when the **Task Selector** selects the next **Task** that the student must solve and places the **Task** into the **Tool** in order to be solved by the student. The four basic methods to do "task

selection" are described in (VanLehn 2006). The **Task Selector** needs to have access to a source of **Tasks**. The **Task Factory** is the source of **Tasks**. It either reads **Tasks** stored in the **Knowledge Base** (reading previously human-authored tasks) or creates tasks in real-time.

- The **Step Analyzer** compares the student's UI behavior to the correct **Steps** of the **Task** and provides that information to the **Assessor** and the **Pedagogical Module**. The **Step Analyzer** typically determines whether the student's step is correct or incorrect, and which of the task's steps most closely matches the student's step. The **Assessor** updates the **Learner Model**. The **Pedagogical Module** provides **Help** using different strategies such as providing immediate or delayed help or providing requested or unsolicited help.
- The **Learner Model** represents the knowledge, difficulties, and misconceptions of the student. The **Learner Model** lists **Tasks** assigned to the student, 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 **Hints** requested and **Feedback** (errors made) is kept. Most importantly, for each **Knowledge Component,** a mastery measure is also kept. The **Task Selector** relies on the **Learner Model's** measures of **Knowledge Component** mastery to choose a **Task** that is neither too hard (too many unmastered knowledge components) nor too easy (too many mastered knowledge components).

Figure 2 extends the ITS structure shown in Figure 1 in order to identify components (functional and data) and their relationships. UML notation is used inside the Core block to create a first attempt of the object-oriented model. The next section shows the details of this model using a pattern-based approach.

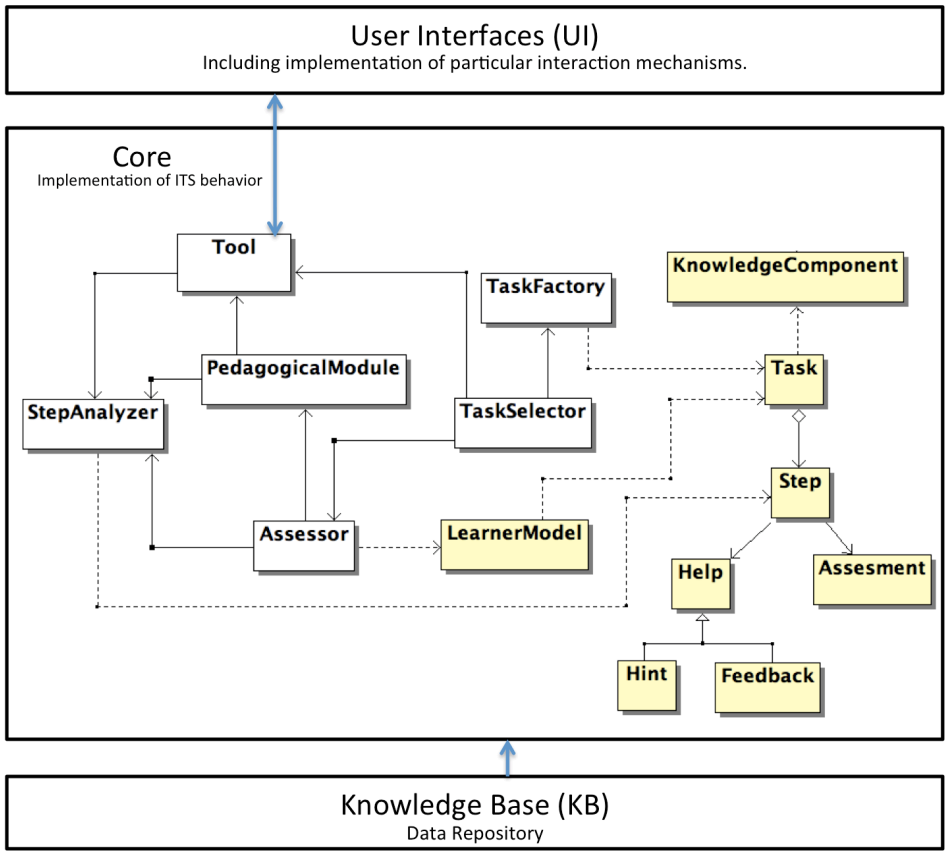

Fig. 2. ITS model. White boxes represent functional components and gray boxes represent data components. Relationships of association, dependency, composition, and inheritance are shown using UML notation.

3.1 Pattern-Based Modeling

The model in Figure 2 represents a conceptual description of who is doing what and corresponds to an abstraction of the expected functionality of each internal component in 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 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; this provided us with a template for the software design and therefore for the implementation. Table 1 shows the relationship between components previously defined matched with 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 the set of ITS subsystems.
TaskSelector	**STRATEGY**	TaskSelector is implemented using **STRATEGY** pattern to deal with the fact that selecting the next Task for the student is done with different algorithms (methodologies) described in (VanLehn 2006).
TaskSelector and Assessor	**OBSERVER**	The relationship between TaskSelector and Assessor can be described by **OBSERVER** pattern. TaskSelector needs information about changes in the LearnerModel (performance of the student) maintained by Assessor, 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, which are the task's steps, and passes the request along the chain until one object handles it. Handling a request means recognizing a student's UI event as a step that is either correct or incorrect.
Assessor	**STRATEGY**	Assessor implements **STRATEGY** to maintain the LearnerModel. Diverse strategies could be tried to store and recover the LearnerModel information. A typical strategy consists of associating each step with a set of knowledge components that represents what the student would need to know in order to get that step correct. When the Assessor is informed that a particular Step is correct, it increments the mastery of the associated knowledge components. On the other hand, if the student got the Step wrong, then Assessor reduces the mastery of the associated knowledge components.
Pedagogical Module	**STRATEGY**	PedagogicalModule 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" (Vygotsky 1978), where tasks are neither boringly easy nor frustratingly difficult, but instead these tasks afford maximal learning and motivating challenges.
Step	**COMPOSITE**	**COMPOSITE** pattern allows us to compose Steps into tree structures to represent part-whole hierarchies. **COMPOSITE** pattern lets us treat individual Steps and hierarchies of Steps (and sub-Steps) uniformly.
Assessor and StepAnalyzer	**OBSERVER**	The relationship between Assessor and StepAnalyzer is 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 PedagogicalModule and StepAnalyzer can be described by **OBSERVER** pattern. PedagogicalModule needs information about student performance in each Step. That information is obtained from StepAnalyzer.

Finding the appropriate pattern to be applied to each component and relationship was a process based on experience and literature research (Buschmann et al. 1996, Gamma et al. 1995). There is no set of rules on how to choose a pattern; instead, a firm knowledge of existing patterns as well as the problems they solve is required in order to effectively use patterns to describe what happens within a given 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 (Buschmann et al. 1996) and (Gamma et al. 1995), we took the keywords observer, abstract factory, builder, chain of responsibilities, strategy, communicator, facade, composite, and singleton; each pattern is fairly close to implementing the task that its name means and what each component is supposed to do; for example, TaskFactory is an **ABSTRACT FACTORY** of Tasks.

3.2 Putting All The 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). Then, our development team would be able to focus on the detailed implementation of the desired functionality, filling in specific places inside of specific files, methods, and attributes (Booch et al. 2007). The UML class diagram for the Core layer is shown in Figure 3.

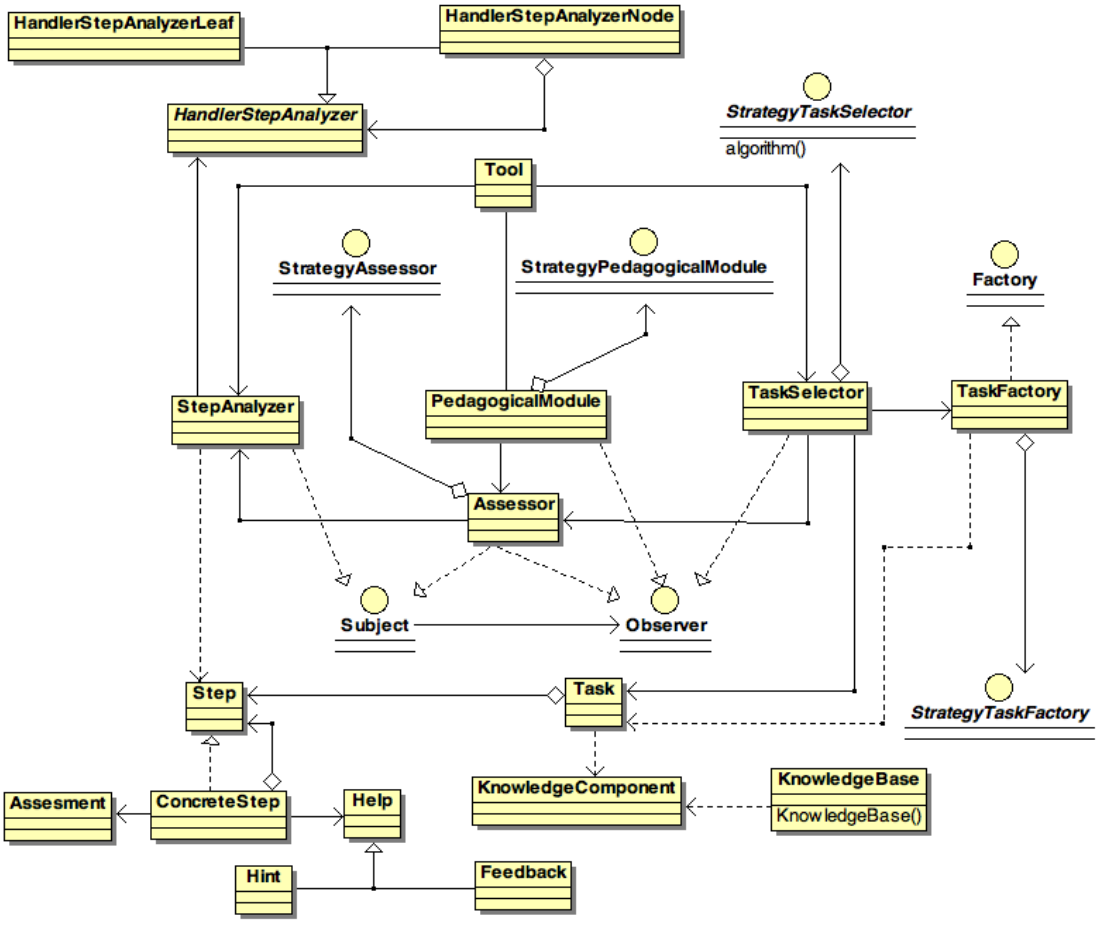

Fig. 3. UML class diagram showing the pattern-based model for the "Core" layer.

It is important to note the following relationships in the diagram shown in Figure 3:

- The implementation of Step as a **COMPOSITE** is highly useful; it provides the capability of managing Steps as one or as a hierarchy of several hierarchical Steps.

- TaskSelector, TaskFactory, Assessor, and PedagogicalModule implement the **STRATEGY** pattern that permit us to define algorithms, encapsulate them, and make them interchangeable. **STRATEGY** pattern lets the algorithm vary independently of the classes that use it. Implementing a new way to select a Task, create a Task, manage the LearnerModel, or provide Help to the student can be done by 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 algorithm method; and (3) create as many additional methods and/or attributes as needed.
- 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.

3.3. Implementation

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 with a shifting development 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.

The implementation of the Strategy classes included:

- StrategyTaskSelector interface implemented in SequencialTaskSelection class. SequencialTaskSelection class defines a strategy that presents Tasks to the student in a predefined sequential order.
- StrategyTaskFactory interface implemented in TaskFromRepository class. TaskFromRepository class defines a strategy that recovers Tasks from text files.
- No strategy implemented for Assessor, this component is still an ongoing part of the project.
- StrategyPedagogicalModule interface implemented in ConditionalPedagogicalStrategy class. ConditionalPedagogicalStrategy class defines a strategy in which, conditionally, the presence of certain events or actions from the student launches pre-established responses.

These classes, which implement Strategy interfaces, are not shown in Figure 3 due to space limitations.

Regarding the Tool component, Tool is a facade for an environment in which the student is able to learn about systems dynamic modeling, using a 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 problem, and then define nodes and edges, enter values or equations in each node, run the model and compare its predictions to given facts. If any of the model's predictions is 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 (VanLehn 2011). Figure 4 shows the implementation of the Tool component. Tool component consists on a Canvas in which a Graph is drawn. A Graph is composed by nodes (Vertexes) 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. Vertexes, Edges, and Graph can be selected from the Canvas and be manipulated (drag and drop, deleted, and so on).

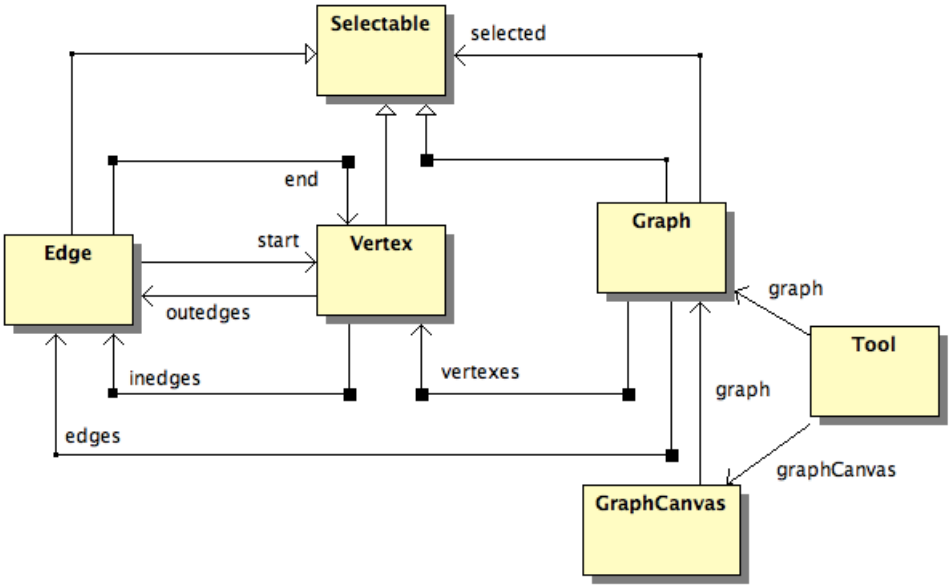

Fig. 4. UML class diagram showing the Tool component encapsulated in the model as Tool.

4. EXPERIENCE REPORT AND EVALUATION

In this section we will describe our experience using design patterns to create an ITS model and implementing it to create the AMT software project.

Stakeholders mentioned the following favorable qualities:

- **Incremental development**. Building the AMT project in an incremental way over the course of three years allowed us to partition the implementation of new functionality into discrete tasks that were worked on independently from others, therefore, reducing the time of deployment adding programmers in particular moments of the project. Incremental development allows us to provide a new functionality or a new version of a current functionality in a window time of two or three weeks.
- **Shifting development team**. Programmers, even without knowledge of patterns, were able to focus their attention on the requirements assigned to them; each programmer worked on completing a specific module or set of components (defined as a pattern section) and relationships between components were almost entirely defined by patterns. We used Subversion (Collins-Sussman 2004) to maintain a common repository of the project as a tool to support branching and merging processes.
- **Communication**. Since diverse stakeholders such as researchers in education technology, computer scientists, developers, and instructional designers were involved, design patterns helped us agree on the structure of the system and communicate it to the programmers for each individual component in the project. 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 disputes emerged with stakeholders regarding the following:

- **Size**. Stakeholders point to the increase in size of code while using patterns as an issue. While using patterns generates more code in our project, this is not only due to patterns (interfaces and abstract classes declarations), but also because we decided to maintain the cyclomatic complexity (McCabe 1976)

for every method under 10, which means applying a "divide and conquer" strategy that generates more methods in the system.

- **Deployment Time**. Since in each iteration our first step focused on the design definition (patterns), software prototypes delayed its appearance. But once the first prototype was presented, new prototypes emerged quicker that in previous projects.

Finally, developers mentioned the following favorable qualities:

- **Collaboration**. Sharing constructions between developers was a key element to counterbalance the effect of a constant shift in the people in the development team.
- **Productivity**. Several prototype versions of the project were created to test new options of functionality as well as new pedagogical approaches.
- **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 a software architecture because they can identify components and relationships among these components inside their systems. However, this does not mean that standard and good practices, such as design patterns, have been followed. We took 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 reusable, extensible, and adaptable. Using design patterns improved our communication, collaboration, and productivity. Design patterns facilitate the knowledge transfer across a highly shifting development 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 patterns 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 the AMT system in three years of work, with a high rate of changes in requirements for the product and a high shifting development 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: the first one will be the inclusion of a module for companions to provide support for the student such as learning companions, affective companions, and teachable agents, and the second one will be the inclusion of meta-tutoring components.

REFERENCES

AMT - Affective Meta Tutor. 2012. Arizona State University. http://amt.asu.edu.

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

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". In *Proceedings of the International Conference on Artificial Intelligence in Education*. IOS Press.

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.

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

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

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

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.

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.

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

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

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

Katz, S., Lesgold, A., Hughes, E., Peters, D., Eggan, G., Gordin, M., and Greenberg., L. 1998. Sherlock 2: An intelligent tutoring system built upon the LRDC Tutor Framework. In C. P. Bloom & R. B. Loftin (Eds.), *Facilitating the development and use of interactive learning environments*. 227- 258.

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

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

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

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

VanLehn, K., Lynch, C., Schultz, K., Shapiro, J. A., Shelby, R. H., Taylor, L., Treacy, D., Weinstein, A., and Wintersgill, M.. 2005. The Andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence in Education*, 15(3), 147-204.

VanLehn, K., Burleson, W., Chavez-Echeagaray, M.E., Christopherson, R., Gonzalez-Sanchez, J., Hastings, J., Hidalgo-Pontet, Y., and Zhang, L.. 2011. The Affective Meta-Tutoring Project: How to motivate students to use effective meta-cognitive strategies. T. Hirashima et al. (Eds.). In *Proceedings of the 19th International Conference on Computers in Education*. Chiang Mai, Thailand: Asia-Pacific Society for Computers in Education.

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