Doğuş Üniversitesi Dergisi, 2002, (5), 109-122

THE UNIFIED SOFTWARE DEVELOPMENT PROCESS AND FRAMEWORK DEVELOPMENT

Abdelaziz Khamis & Ashraf Abdelmonem
Department of Computer Science, Institute of Statistics, Cairo University

ABSTRACT: Application frameworks are a very promising software reuse technology. The development of application frameworks is a complex process. Many methodologies and approaches have been proposed with the purpose of minimizing the complexities. The Unified Software Development Process directly addresses the complexity challenge of today's software applications. In this paper, we explore the role of the Unified Software Development Process together with a popular CASE tool: *Rational Rose*, in managing the complexity of developing application frameworks.

Keywords: *Application Frameworks, The Unified Software Development Process, Model Tracing Tutors, Rational Rose.*

ÖZET: Uygulama iskelet yapıları çok umut vaadeden bir yeniden kullanım yazılım teknolojisidir. Uygulama iskelet yapılarının geliştirilmesi karmaşık bir süreci gerektirir. Bu karmaşık süreci minimize etmek amacıyla birçok yöntem ve yaklaşım önerilmiştir. Birleşik Yazılım Geliştirme Süreci bugünkü yazılım uygulamalarındaki karmaşıklık engelini direkt olarak hedefleyen bir yöntemdir. Bu yazıda uygulama iskelet yapılarındaki karmaşıklığın yönetimi konusunda Birleşik Yazılım Geliştirme Süreci'nin rolü yaygın olarak kullanılan bir CASE (Bilgisayar Destekli Yazılım Mühendisliği) aracı olan "Rational Rose" ile birlikte araştırılmaya çalışılmıştır.

Anahtar Kelimeler: *Uygulama Geliştirme Yapıları, Birleşik Yazılım Geliştirme Süreci, Model İzleme Öğretmeni, Rational Rose.*

1. Introduction

The reuse of software components is recognized as an important way to increase productivity in software development (Jacobson, Griss and Jonsson, 1997). Application frameworks are a very promising software reuse technology. The concept of frameworks makes it possible to reuse not just code, but also analysis and design (Fayad, Schmidt and Johnson, 1999; Fayad and Schmidt, 1997).

Developing application frameworks is a complex process. Many methodologies and approaches have been proposed with the purpose of minimizing the complexities of framework development (Fayad, Schmidt and Johnson, 1999; Framework..., 1997). In this paper we have focused on a very promising approach namely, the *framework layering approach.*

The Unified Software Development Process directly addresses the complexity challenge of today's software applications (Jacobson, Booch and Rumbaugh, 1999). The Unified Process is the end product of three decades of development and practical use. Its development has been guided by three leading figures in software development: Jacobson, Booch, and Rumbaugh. In this paper, we explore the role of

the Unified Software Development Process together with a popular CASE tool: Rational Rose, in managing the complexity of developing frameworks.

The paper is organized as follows. The next section overviews the framework layering approach. In section 3, we give a briefly describe the lifecycle of the Unified Software Development Process. The chosen domain for our case study: *Intelligent Tutoring Systems* (ITS) is defined in section 4. The CASE tool: *Rational Rose* is introduced in section 5. Section 6 contains the case study: a working example of framework development using the Unified Software Development Process and Rational Rose. Finally, section 7 includes the conclusion and the lessons learned.

2. Framework Development Approaches

There are many different approaches for framework development. These include: systematic generalization, hot-spot-driven development, and framework layering. The details of these approaches exist in *part five* of (Fayad, Schmidt and Johnson, 1999). However for the purpose of this paper we overview the framework layering approach.

2.1 Framework Layering Approach

Framework layering must be rooted in the application domain in order to meet business needs (Framework..., 1997). Therefore, before discussing framework layering, we identify some application domain concepts.

- *Business sections.* The services offered by any business enterprise can be divided into different areas of responsibility. These areas are called business sections. In a bank, *teller*, *loan*, and *investment* are examples of business sections.
- *Workplace contexts.* The division into business sections is often supplemented by so-called service centers. Service centers can perform most of the common tasks encountered in the various business sections. These centers are referred to as workplace contexts. In a bank, *customer service center*, *teller service*, and *automatic teller machines* are examples of workplace contexts.
- *Business domain.* The core or common parts of the concepts and terms, that is essential to running the business as a whole, is called the business domain. In a bank, *account*, *customer*, and *interest rate* are examples of business domain concepts.

Thus, frameworks should be organized along business-domain, business-section, and workplace-context layers. We now go on to look at the basic functionality of each layer and the relations between them.

The Business Domain Layer

This layer contains the core concepts for the business as a whole. It thus forms the basis for every application system in this domain. It is crucial to make an

appropriate division or separation between the part of a core concept that belongs to the business domain and the parts belonging to the business sections.

The Business Section Layers

These layers consist of frameworks with specific classes for each business section. The frameworks in these sections are based on the Business Domain Layer. To relate the core concepts of the Business Domain Layer to their extensions in the Business Section Layers the Role Object Pattern has been developed.

The Role Object Pattern is applied to make one logical object span one or more layers. The core object, which resides in the Business Domain Layer, is extended by role objects, which reside in the Business Section Layers. A role is a client-specific view of an object playing that role. One object may play several roles, and different objects can play the same role.

The Application Layers

These layers provide the software support for the different workplace contexts. The separation of the Application Layers from the Business Section Layers is motivated by the need to configure application systems corresponding to different workplaces.

3. The Unified Software Development Process

A software development *process* is the set of activities needed to transform a user's requirements into a software system. However, the real distinguishing aspects of the *Unified Process* are captured in the three key words – *use-case driven, architecture-centric,* and *iterative and incremental* (Jacobson, Booch and Rumbaugh, 1999). In this section we will describe these three key words. Then, we will give a brief overview of the lifecycle of the Unified Process.

The Unified Process is Use-Case Driven

A use-case is a piece of functionality in the system that gives a user a result of value. Use cases capture functional requirements. All the use-cases together make up the *use-case model,* which describes the complete functionality of the system. However, use cases are not just a tool for specifying the requirements of a system. They also derive its design, implementation, and test; that is *they derive the development process.* Based on the use-case model, developers create a series of design and implementation models that realize the use cases. The testers test the implementation to ensure that the components of the implementation model correctly implement the use cases.

The Unified Process is Architecture-Centric

Every software product has both a function and form. The form in software is the architecture. The use cases define the function, and the architecture defines how functionality is related and integrated. While it is true that use cases drive the development process, they are not selected in isolation. They are developed in cycle with the *system architecture.* That is, the use cases drive system architecture and the system architecture influences the selection of the use cases. The role of software

architecture is similar in nature to the role of architecture plays in building construction. The builder needs to consider not only the function of the building but also its form.

The Unified Process is Iterative and Incremental

In an iterative and incremental lifecycle, development proceeds as a series of iterations that evolve into the final system. Each of those iterations consists of the following process components: requirements, analysis, design, implementation, and test. The developers do not assume that all requirements are known at the beginning of the lifecycle; indeed change is anticipated throughout all phases.

3.1 The Lifecycle of the Unified Process

The lifecycle of the Unified Process repeats through four main incremental phases:

- *Inception* - top-level abstract use-cases are developed.
- *Elaboration* - the software product's use cases are more fully developed and an architectural view of the system is produced.
- *Construction* - the software product is built.
- *Transition* - the software product is moved from testing to development.

Each phase is composed of one or more iterations. Each iteration consists of five core workflows: *requirements*, *analysis*, *design*, *implementation*, and *test*. The number of iterations needed depends on the size and complexity of the software system to develop.

4. Intelligent Tutoring Systems (ITS)

Intelligent tutoring systems allow the emulation of human teacher in the sense that an ITS can know what to teach (*domain content*), how to teach it (*instructional strategies*), and learn certain relevant information about the student being taught. This requires the representation of a domain expert's knowledge (called the *Expert Model*), an instructor's knowledge (called *Instructional Model*), and the particular student being taught (called the *Student Model*). Through the interaction of these models, ITSs are able to make judgments about what the student knows and how well the student is progressing. The *Instructional Model* can then automatically tailor instruction to the student's needs (Murray, 2001).

The most common type of ITSs teaches procedure skills, the goal is for students to learn how to perform a particular task. These tutors are referred to as ***model-tracing tutors*** because they contain an expert model that is used to trace the student's responses to ensure that these responses are part of an acceptable solution path. Examples of such tutors exist in (Corbett and Anderson, 1992; Lewis, Milson and Anderson, 1987). An architectural view of model-tracing tutors may be produced as the one given in Figure 1.

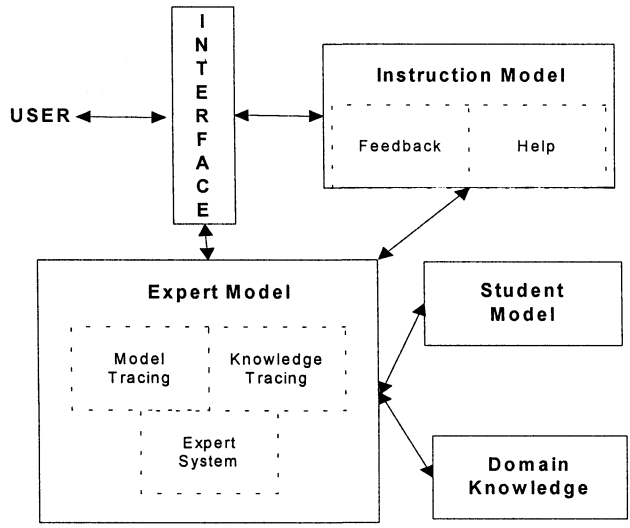

Figure 1. An Architectural View of Model-Tracing Tutors

Model-tracing tutors embodied a number of key features about how computer-based instruction should be realized. These features include (Cognitive Tutors ..., 2001):

Expert Model: What forms the backbone of model-tracing tutors is an expert model, realized as a set of *production rules*, that contains all the knowledge needed to solve problems within the domain being tutored. The model is capable of generating a set of production sequences that represent correct solutions of the problem. Correct (*On-path*) actions on the student's part are recognized if they are along one of the correct solution paths generated by the model. If the student is correct, the tutor does not comment but rather allows the student to progress with the solution. If the student performs a wrong *(off-path)* action, instruction is focused on getting the student back on path.

Student Model: The assessment by the tutor of the student's current knowledge state constitutes the so-called *student model.* As the student interacts with the system, getting some of the answers right and others wrong, this student model is updated. The system contains a list of skills that make up the tutorial domain. Skills correspond to a sequence of production rules that result in a student action. Each skill is considered to be in either a learned or unlearned state, with a probability assigned to it that it is currently in the learned state.

Student's Interface: The student's interface must vary across the different tutored domains. The interface that the student interacts with in the geometry tutor is very much like a typical computer drawing program, whereas the interface used by programming tutors is a structured text editor. However, no matter the specifics of an interface, for each student action within the interface (clicking a button, typing in a text field, selecting a menu option, etc), that action can be checked against the expert model. This process of checking student actions using the expert model is referred to as model tracing.

Error Feedback and Help: The tutors possess two types of instruction. If the student makes a recognizable error (a bug), a message can be given explaining why it is an error. This is generated from a buggy production that embodies the error. If the student asks for help, a help message is presented to guide the student to the correct solution. This message is generated from the information along a correct path.

5. Rational Rose

Rational rose is a graphical software engineering tool that supports object-oriented analysis, design, and implementation. It supports the Unified Software Development Process and the Unified Modeling Language (UML) (Boggs and Boggs, 1999; Quatrani, 2000). Rational Rose may be used to create the following diagrams:

- *Use-Case* diagrams.
- *Class* diagrams.
- *Collaboration* diagrams.
- *Sequence* diagrams.
- *Component* diagrams.
- *State-Chart* diagrams.
- *Deployment* diagrams.

6. Case Study: An ITS Framework

The purpose of the case study is to provide a working example of framework development using Rational tools. The chosen domain was *Intelligent Tutoring Systems* (ITS), which is a domain whose concepts are well known to most people.

6.1. Inception Phase

6.1.1 Capture the Requirements

The end result of this workflow is a tentative use-case model. To build such a model we first develop a business model in two steps:

1. Prepare a business use-case model that identifies the actors to the business and the business use cases that actors use. The business use-case model for ITS framework is shown in figure 2.

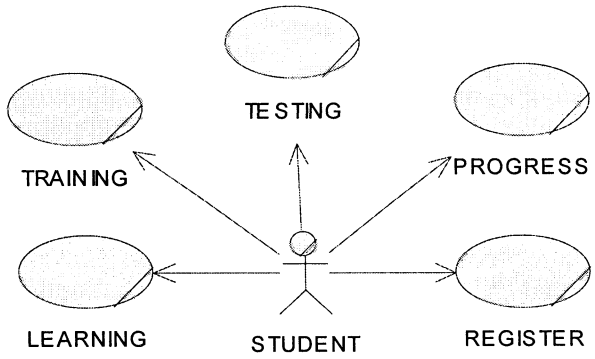

Figure 2. Business Use-Case Model

2. Develop a business object model consisting of workers, business entities, and work units that together realizes the business use cases. Figure 3, shows the business object model for ITS framework.

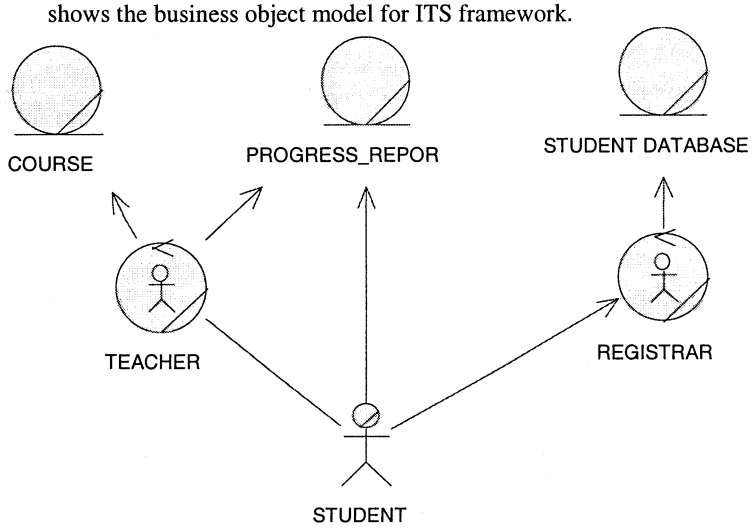

Figure 3. Business Object Model

Using the business model as input, we can create a tentative use-case model in two steps: First, we identify an actor for every worker and business actor. Second, we create a use case for each role of an identified actor. Figure 4, shows a tentative use-case model for ITS framework.

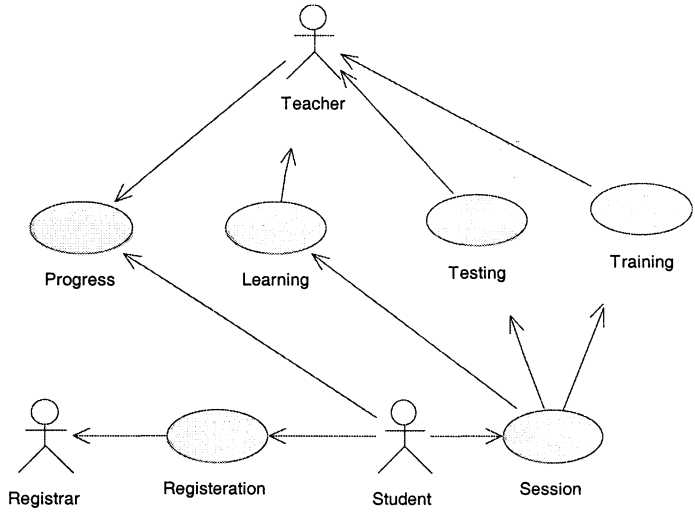

Figure 4. A Tentative Use Case Model

During the inception phase, the flow of events for the most important use cases is documented as state-chart diagrams. A state-chart diagram for the *Learning* use-case is shown in Figure 5. It shows how the learning use case moves over several states in series of state transitions.

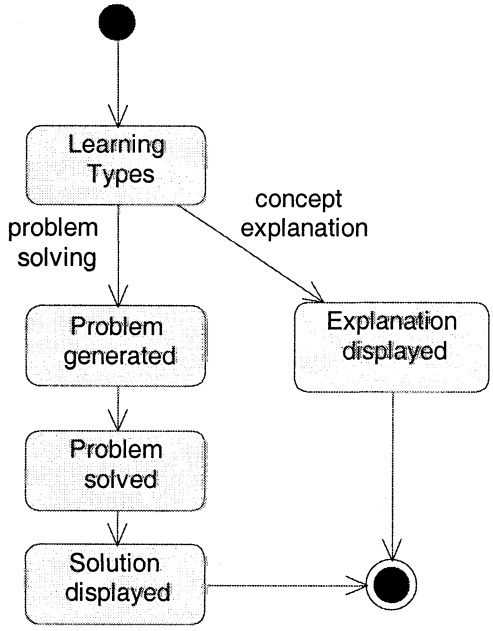

Figure 5. The State-Chart Diagram for the Learning Use-Case

6.1.2. Analysis

The result of the analysis workflow is an initial analysis model. To build this model, we identify analysis packages by allocating a number of related use cases to a specific package and then realize the corresponding functionality within that package by identifying the *control*, *entity*, and *boundary* classes. Figure 6, shows an initial analysis model for ITS applications. The realization of the *Session* package is shown in figure 7.

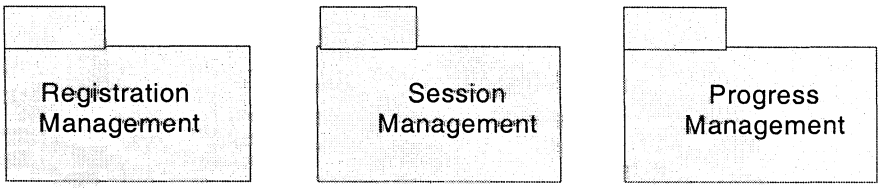

Figure 6. An Initial Analysis Model

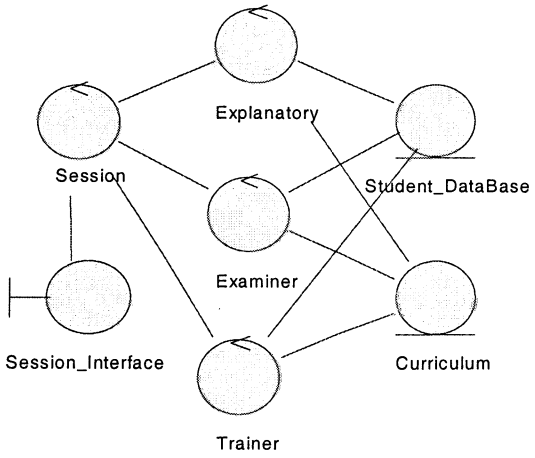

Figure 7. The Session Management Package Realization

After we have an outline of the analysis classes needed to realize the most important use-cases, we describe how their corresponding analysis objects interact using collaboration diagrams. The collaboration diagram for the *Training use-case* is shown in figure 8. Collaboration diagram contains the participating actor instances, analysis objects, and their links.

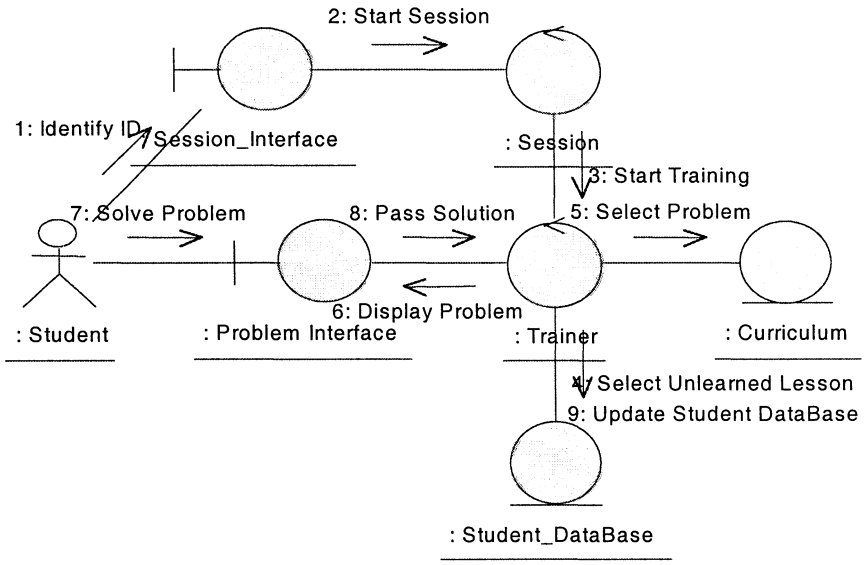

Figure 8. A Collaboration Diagram for the Training Use-Case

6.2 Elaboration Phase

6.2.1. Capture the Requirements

In this workflow, we identify additional use cases; beyond those identified in the inception phase. Then, we describe those use cases using state-chart diagrams. Examples of additional use cases include: problem selection, problem solving, example selection, example solving, test generation, adding student, and deleting student.

6.2.2. Analysis

In this workflow, we identify additional analysis packages; beyond those identified in the inception phase. Then, we realize the corresponding functionality within those packages by identifying the control, entity, and boundary classes. Examples of additional packages include: student-database management, and curriculum management.

6.2.3. Design

Design is in focus during the end of elaboration and the beginning of construction phases. It contributes to a sound and stable architecture and creates a blueprint for the implementation model. Figure 9 shows a layered architecture for ITS applications. The given architecture contains three categories of layers namely, the *ITS Application Layers*, the *ITS Section Layers*, and *the ITS Domain Layer.*

The ITS Application Layers provide the software support for the different workplace contexts. The ITS Section Layers consist frameworks with specific classes for each

business in ITS namely, training, teaching, testing. Finally, The ITS Domain Layer contains the core concepts for the business of ITS as a whole.

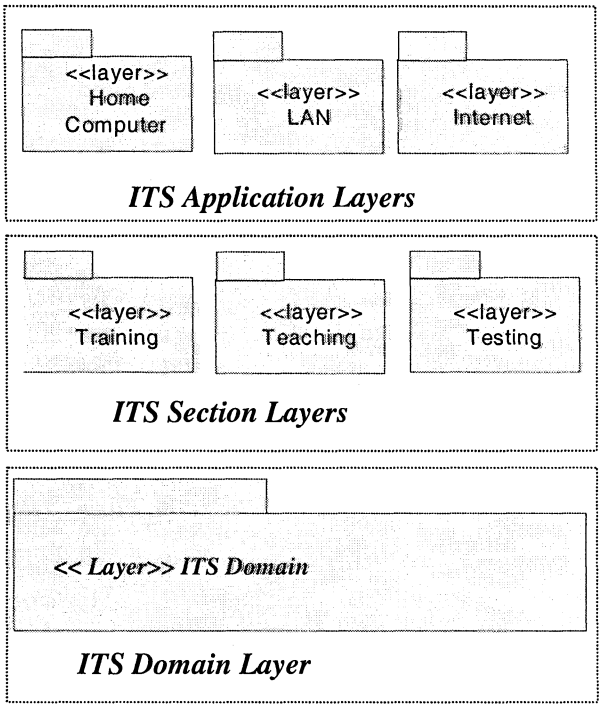

Figure 9. Design Model Layers

Based on the packages in the analysis model, we identify corresponding subsystems to be included in the design model. Figure 10 shows some design subsystems that are based on existing analysis packages.

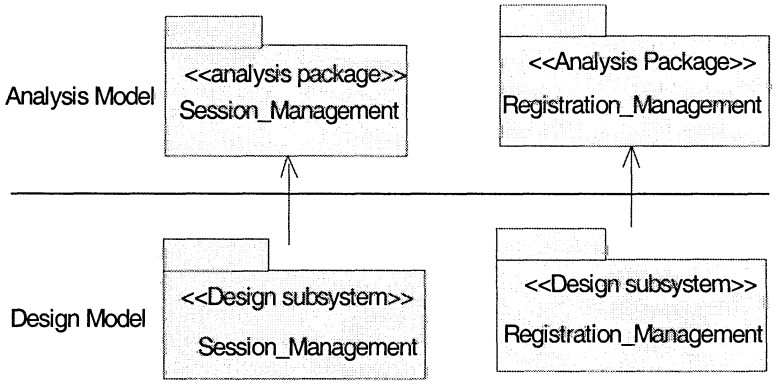

Figure 10. Some Design Subsystems Based On Existing Analysis Packages

Now, we identify the design classes that trace to the analysis classes in the analysis model. Figure 10 illustrates the design classes that trace to Session interface and Session analysis classes.

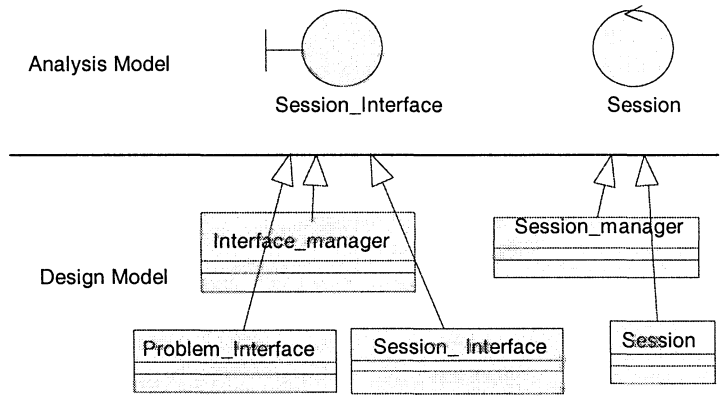

Figure 10. Design classes in the design model tracing to analysis classes in the analysis model

At the end of the elaboration phase, we create the design classes, identify the operations, and describe those operations using the syntax of a programming language. Figure 11 shows the operations and attributes for some classes.

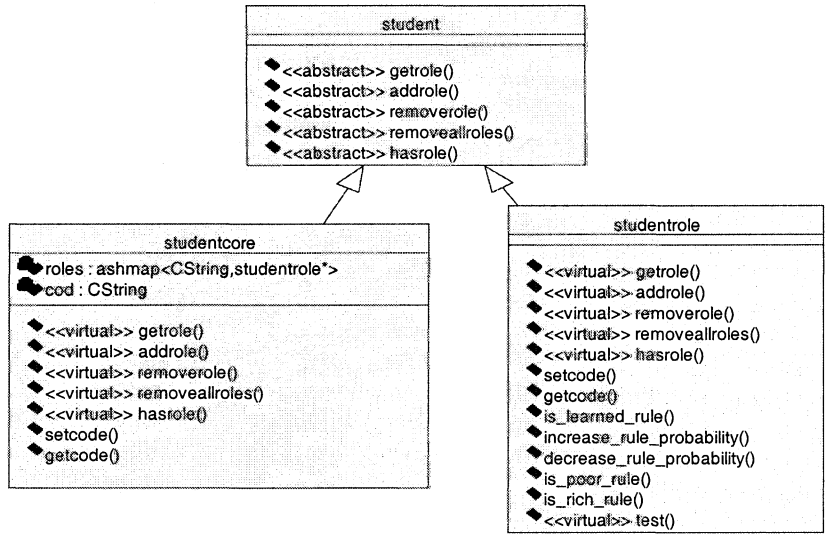

Figure 11. The Student, StudentScore, and StudentRole classes

6.3. Construction Phase

6.3.1. Implementation

The result of the implementation is an implementation model that describes how elements in the design model, such as design classes, are implemented in terms of components such as source code files, executables, and so on. Figure 12 illustrates the most important components in our implementation model.

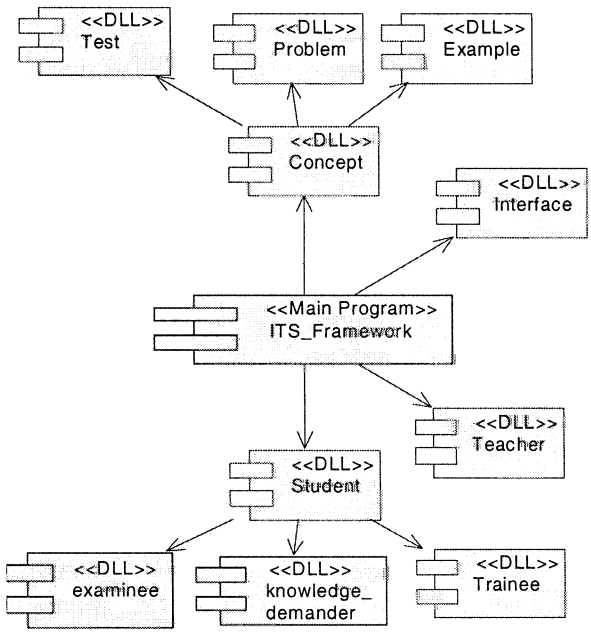

Figure 12. An Implementation Model

7. Conclusion

Although a large number of successful frameworks have been developed during the last several years, designing a high-quality framework is still a difficult and complex task. Therefore, as demand for frameworks increases, there is a need to use the right software development process and the right CASE tools that help in managing their development. Our objective in this paper was to explore the role of the Unified Software Development Process and Rational Rose in managing the complexity of framework development.

The main claims of this paper are:

- The framework development time may be reduced considerably by using Rational Rose. We usually spend hours trying to, modify our system's

design to account for last-minute requirements changes, generate our system's code, and track down memory-related errors in our code. Modeling, code generation, unit testing, and configuration management handle these problems, so it saves a lot of programmer's time and thinking.

- Nothing is ever right the first time. For this reason, iteration plays such a prominent role in the Unified Software development Process. The key point to iteration is dealing with feedback. Feedback from users helps shape the evolution of the requirements. Feedback from developers validates the architectural structure and evolves the object model into a realizable form.

- The Rational Rose CASE tool may be used to automate some aspects of the Unified Software development Process. Thus, using Rational Rose will lead to cost-effective realization of application frameworks.

- The Rational Rose CASE tool provides comprehensive support to the software development process. Quality assurance is an integral part of this process through templates for guidelines and many checkpoints to check the quality of artifacts to be delivered in each phase.

REFERENCES

BOGGS, W., BOGGS, M. (1999). *Mastering UML with Rational Rose*, Financial Times Management, London.

Cognitive Tutors: Lessons Learned, (2001). J. ANDERSON (et al.) http://sands.psy.cmu.edu/ACT/papers/Lessons_Learned.html.

CORBETT A., ANDERSON, J. (1992). "LISP Intelligent Tutoring System: Research in Skill Acquisition", in *Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches*", Edited by J. LARKIN and R. CHABAY, Publishers: Lawrence Erlbaum Associates.

FAYAD, M., SCHMIDT, D., JOHNSON, R. (1999). *Application Frameworks, in Building Application Frameworks*, John Wiley & Sons.

FAYAD, M., SCHMIDT, D., JOHNSON, R. (1999). *Building Application Frameworks*, John Wiley & Sons.

FAYAD, M., SCHMIDT, D. (1997). "Object-Oriented Application Frameworks", *Comm. of the ACM*, Vol. 40, No. 10.

Framework Development for Large Systems. (1997). D. BÄUMER (et al.). *Comm. of the ACM*, Vol. 40, No. 10.

JACOBSON, I., BOOCH, G., RUMBAUGH, J. (1999). *The Unified Software Development Process*, Addison-Wesley.

JACOBSON, I., GRISS, M., JONSSON, P. (1997). *Software Reuse: Architecture, Process and Organization for Business Success*, Addison Wesley.

LEWIS, M., MILSON, R., ANDERSON, J. (1987). "The teacher's Apprentice: Designing an Intelligent Authoring System for High School Mathematics", in *Artificial Intelligence and Instruction: Applications and Methods*, Edited by G. KEARSLEY, Addison-Wesley Publishing Company.

MURRAY, T. "Authoring Knowledge Based Tutors", http://www.ac.unmass.edu/~tmurray/papers/JLSEon/JLS96.html.

QUATRANI, T. (2000). *Visual Modeling with Rational Rose 2000 and UML*, Addison-Wesley.