STUDIES IN LOGIC, GRAMMAR AND RHETORIC 25 (38) 2011

Clinical department information system's internal structure

Piotr Ziniewicz[1], **Paweł Malinowski**[1], **Robert Milewski**[1], **Stanisław Zenon Mnich**[1], **Sławomir Wołczyński**[2]

[1] Department of Statistics and Medical Informatics, Medical University of Bialystok

[2] Department of Reproduction and Gynecological Endocrinology, Medical University of Bialystok

Abstract. The construction of an advanced information system supporting the work of a clinic is a great challenge. Particularly, after the initial determination of the functionality of the system, its internal structure must be designed. For the major elements of the system their interaction must be also accurately determined.

Introduction

In the middle of the 20th century the first modern general-purpose computing machines were created. The first computers were much more bulky than those used today and had a low computational speed. Since then, computers underwent tremendous, almost exponential evolution. With the new capabilities of hardware its software is also subject to the process of development. In today's world the computer has become a tool for everyday use. The ongoing process of computerization and the growth of user requirements force the development of information systems of increasing complexity.

Many modern fields of human activity, including medicine, benefit from the ongoing process of computerization. Modern hospital information systems (HIS) [2] are the example of this. The task of the described here JeNaK system (from Polish "Jednostka Naukowo-Kliniczna" – "Clinical Science Unit") is a comprehensive management of the medical university's selected clinical unit. Creation of such a complex system poses a great challenge, therefore, during work on each of the elements of the system, modern software engineering tools, including UML modeling language, were used. The aim of this article is to describe the internal structure of the proposed system. Selected class diagrams of key system elements will be described in detail.

Application in medicine and engineering systems

Informatics systems engineering took shape thanks to the development of hardware capabilities which induce the evolution and growth of software complexity. It was quickly noted that the creation of such systems, possible thanks to the work of many programmers, but without an adequate methodological background, often leads to failure [5]. This led to the emergence of software engineering. Standardization of the software creation process had found its culmination in the establishment of the Universal Modeling Language (UML). UML version 2.2 [1] describes informatics system by means of 14 diagrams representing various aspects of the modeled system.

With the software development, first systems, supporting the work of hospitals emerged. They were used for cost accounting and hospital management by *stricte* economic mean [6]. Since then, these systems have undergone a profound evolution. Most modern HIS focuses its attention around medical and organizational information flow, crucial to hospital operation as a therapeutic unit. They allow for e.g. to store patient information, his/her medical history, or treatments. Nowadays, there are many hospital information systems. Most of them supports the standard methods for exchanging information such as DICOM [4] or HL7 [3].

Despite the diversity of available HIS, there is no system so far that would support the clinical aspect of the specified hospital unit. In contrast to the typical HIS, the JeNaK system not only assists the work of doctors and hospital, but also takes into account the presence of scientists, teachers and technical workers of the clinic, and also taught students. Due to the nature of the clinical unit, the JeNaK system includes collection, storage and processing of various data relevant to this unit type.

JeNaK system structure

After a detailed analysis of potential user's requirements through the use case diagrams [8], the next step is to build class diagrams – the basic components of the system. These diagrams describe the internal, static fabric of the proposed system. Actual structure of the clinical unit, internal dependencies, relationships and functions should be directly reflected in classes.

Concepts of class and object are key issues in modern software engineering. They define the paradigm of object-oriented programming (OOP), with its many consequences. The class, often identified simply as a type, is a description, which allows creating its representation (instances) – objects, that exists in the computer machine memory. Class binds information related to the current state of an object with its functionality, which allows for internally consistent change of that state. The state is described by the so-called class fields and functionality through the methods.

Typically, most of the fields and class methods that can be used need an object to operate on it. Most classes have special methods that are executed when an instance is created or is destroyed. These special methods are the constructor (in C++ it has the same name as class) and the destructor (in C++ it has name of the class presided with tilde). There are usually a number of constructors, but only one destructor is defined. In addition, some classes have fields or methods that are shared by all objects of the class, without affecting their internal state. They are called static fields/methods. To read/execute them, an object is not needed, only the class itself. UML represents a class with a rectangle divided into three smaller sections by horizontal lines [Fig. 1]. Upper section defines class name, the middle – all fields of the class, the bottom – all of its methods. Depending on the level of detail, a description of the types of fields and types of values returned by the method (after the colon) and the arguments of the method (as a comma separated list, each argument is described by its name and type after the colon) can be presented as well. Static methods/fields are underlined.

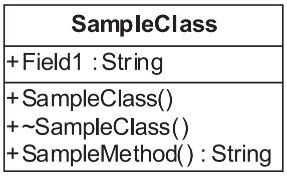

Fig. 1. Simple class diagram

In addition to the class concept, an important part of OOP is data encapsulation (hiding). Typically, the internal state of the object and part of its functionality should be protected from outside access. This makes the management of information flow within the program significantly easier. Three types of access specifications [Fig. 2] are defined (UML designation in parentheses):

- public (+) – a method or field is available for other objects from the outside
- protected (#) – a method or field is available for objects whose class is derived (inherits) from the current class

- private (–) – a method or field is available only to the methods of the class.

The C++ language, which was chosen by the authors as the target programming language of the JeNaK system, allows for the weakening of this mechanism, when it is preferred due to the readability of the code and its performance. A class can grant access to protected or private methods to another class by declaring it as a "friend" [Fig. 2].

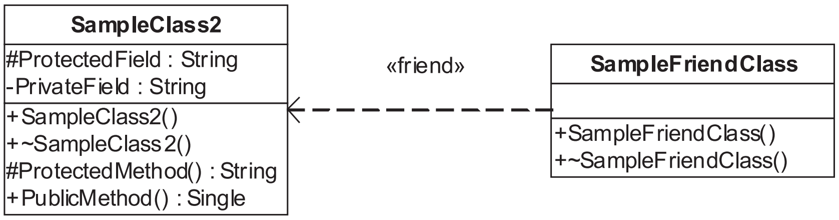

Fig. 2. Simple class diagram with friend and different function/members kinds

Another important OOP issue is interdependence of classes and objects. This relationship is (usually) a logical consequence of the real dependence of the modeled system. Dependence at the class level (the fact of inheritance), and at the object level (relation or inclusion) is distinguished.

Inheritance – "is a" dependency – occurs in arrangement of two classes, when one of them – subclass – is extended or specialized form of another – a super class [Fig. 3a]. Languages supporting OOP allow to treat each child class object (the car in [Fig. 3a]) as the parent class object (vehicle in [Fig. 3a]) – this is polymorphism mechanism. The above mentioned specialization may involve redefining some methods, or changeing their action in relation to the original class. Borderline case of such dependency are the so-called abstract classes, which represent only the method names (that is, the expected functionality), without defining their actions. Instances of such classes cannot be created, but objects, which class derives from it, can be treated as representatives of these classes, according to the mechanism of polymorphism. C++ allows the inheritance from multiple base classes. The fact of inheritance is depicted on class diagram by an arrow with a blank tip directed from the sub class to a super class. The name of an abstract class is written in *italics*.

A link represents a loose association between objects [Fig. 3b]. Usually, it is a functional relationship. A more specialized form of link is inclusion. Two kinds of objects inclusion – composition and aggregation can be listed. Aggregation [Fig. 3c] occurs when an object holds references to other objects, but the lifetime of those objects is independent. Composi-

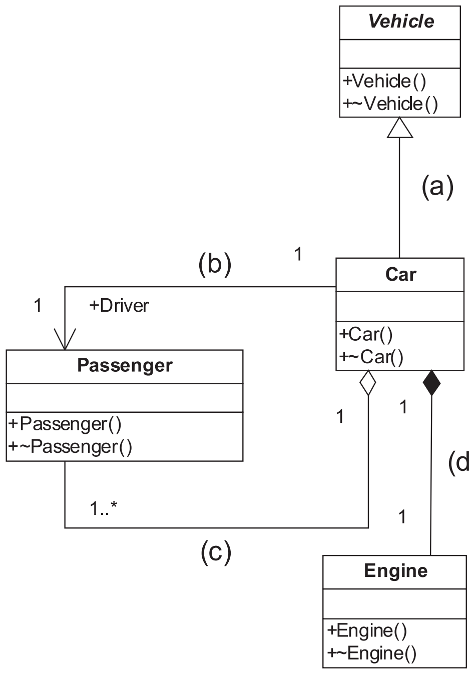

Fig. 3. (a) Inheritance (b) Dependence (c) Aggregation (d) Composition Example

tion [Fig. 3d] is inclusion with a strong dependence of the lifetime of the objects. This means that with the destruction of the container object, contained instances are also destroyed.

The link between objects is represented as a line (or arrow) connecting class boxes. If necessary, higher order associations (not binary) can be drawn with more than two ends by connecting all lines to the central diamond. If named, the line can be adorned with a brief description of relation. The aggregation is determined by adding the hollow diamond-like sign at the container-class end of the line. The composition is similar, but the sign is filled. In case of aggregation and composition, usually quantities of associated objects are determined. Numbers inserted near description of a class means a count of instance occurrence. Frequencies are presented in the following forms (n and m are numbers, $n \leq m$):

- $n..m$ – opposite object contain from n to m instances of this object
- $n..*$ – opposite object contain n or more instances of this object
- n – opposite object contain exactly n instances of this object
- * – opposite object contain 0 or more instances of this object (same as 0..*)

As it has been already mentioned, the class diagram translates real relationships and dependencies to the internal structure of the system. There are many rules for handling such translation. One of the easiest is to treat each noun within the use case diagram as a potential class, and a verb as a method of this class. Class diagrams presented below were developed after a detailed analysis of the system use cases.

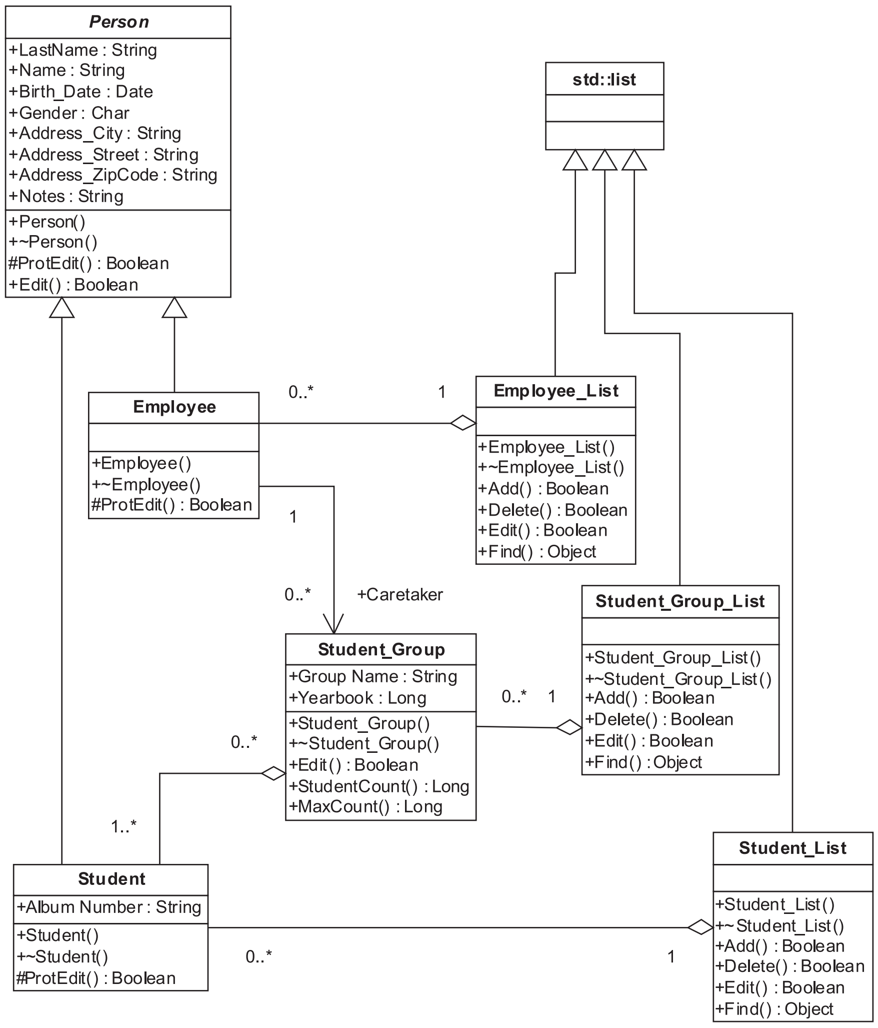

Fig. 4. Class diagram of list containers

On the first diagram [Fig. 4] the structure of the list containers is presented. Abstract class *Person* is created to store basic personal data of students and employers. Two specialized classes – *Student* and *Employee* inherit from it. In addition, the *Student_Group* class, which stores the information about the student-in-group organization is presented. All of these 3 classes have their list containers (*Employee_List*, *Student_List* and *Student_Group_List*) which are used to manage the whole sets of it. These classes are derived from one standard C++ class named std::list. Worth noticing is the fact that each student can belong to many groups. This situation is necessary to reflect that a student can be a participant of many courses (including the elected ones) that can spread across many years of study.

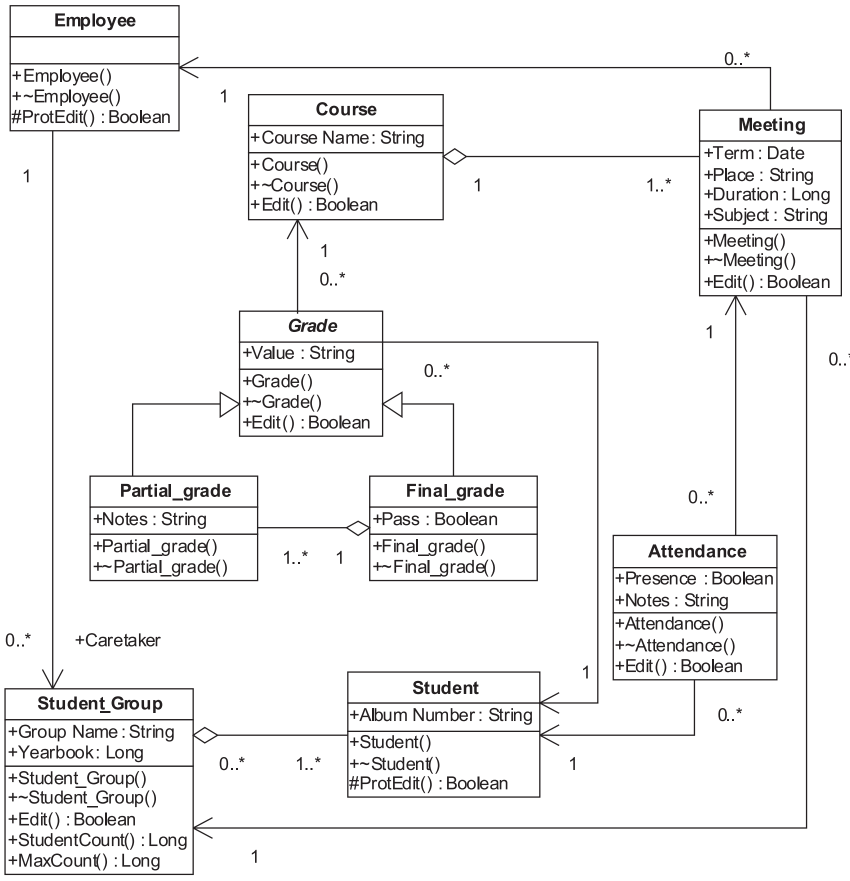

Fig. 5. Class diagram of student evaluation data

Diagram presented on [Fig. 5] shows the dependencies between classes responsible for the storage of student evaluation data. Students participate in courses organized in groups. Each course consists of many meetings during which course material is presented to students. The presence of each student at each meetings is recorded by means of a special class called

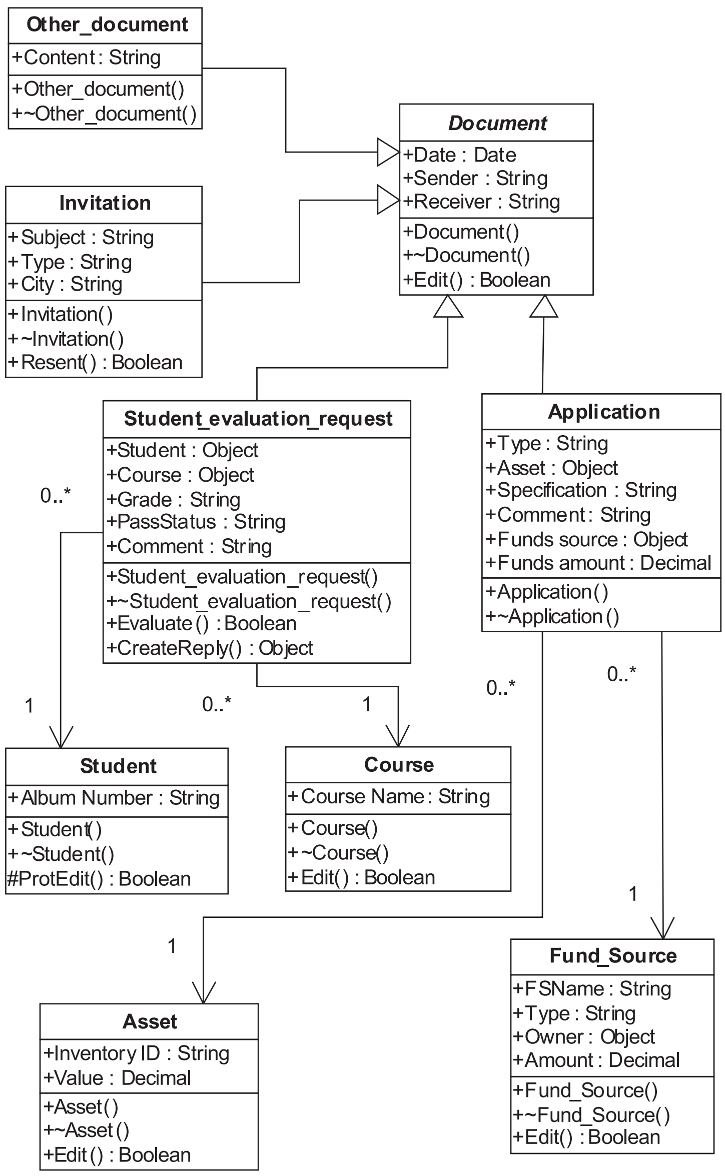

Fig. 6. Class diagram of the mail document derived classes

Attendance. Abstract class *Grade* is used by derived classes *Partial_Grade* and *Final_Grade* to hold respectively partial and final evaluation results. Every grade is related to a course and a student which shows which of them it concerns. The final grade consists of many partial grades and contains additional field that determines if the student passed the course or not.

[Fig. 6] presents many particular classes that derive from one abstract class *Document*. This class is used to support any ingoing and outgoing paper mail documents. Most specialized classes that inherit from it is the *Student_Evaluation_Request* which can be used to obtain, handle and prepare a replay for requests about particular students' results. A pecial method called *Evaluate* can be used to provide adequate reply data and comment to the document. Another method, called the *CreateReply* is able to create a new document containing the reply for the request and fill it with provided earlier data. The class itself is related to the Student and Course classes which enable to point particular student and course which relate to the question.

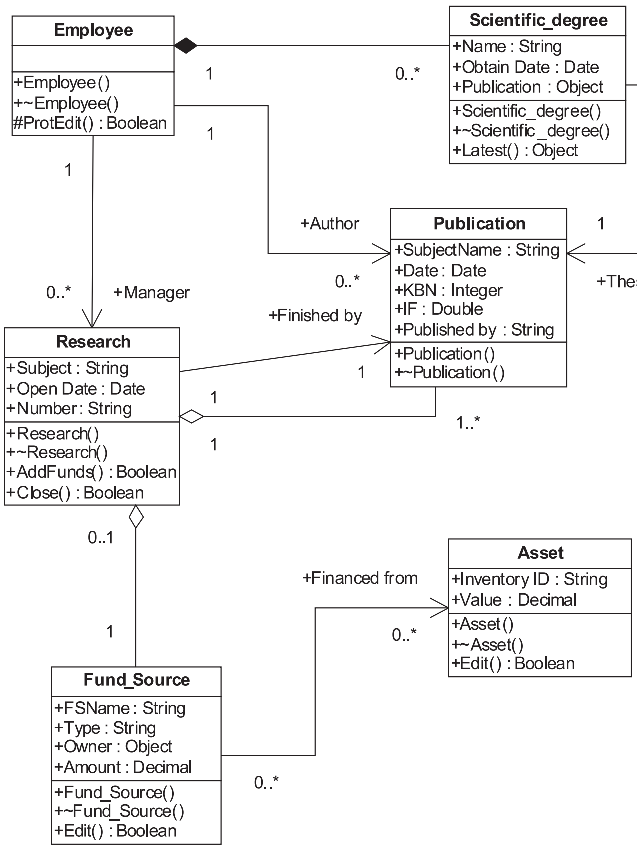

Fig. 7. Class diagram of scientific data

Last diagram [Fig. 7] shows the structure used to store the scientific achievement data of each employee. The composition between *Employee* and *Scientific_degree* classes perfectly reflects the life-time dependency that forces all degrees to be deleted in case of employee deletion. Any publication can be linked to the research but only one can be used as a settlement and finish it.

Conclusions

The JeNaK system was designed to supplement the typical HIS with the administrative, scientific and didactic aspects of the clinical unit. In the paper, a technical draft of selected modules of the system was outlined. Efforts have been made to make it compatible to the user's requirements described in previous articles [7–8]. Further research is required to create class diagrams for the communication with HIS using standard DICOM and HL7 protocols module. In the longer term, ongoing work will focus on the implementation of the system and its deployment in selected clinical units of the Medical University of Bialystok.

REFERENCES

[1] Alhir S. S., Guide To Applying The UML, New York, Springer-Verlag, 2007.

[2] van Bemmel J. H., Mused M. A., Handbook of Medical Informatics, Berlin, Springer-Verlag, 1997.

[3] Benson T., Principles of Health Interoperability HL7 and SNOMED, London, Springer-Verlag, 2010.

[4] Pianykh O. S., Digital Imaging and Communications in Medicine. A Practical Introduction and Survival Guide, Berlin, Springer-Verlag, 2008.

[5] Sommerville I., Inżynieria oprogramowania, Warszawa, Wydawnictwa Naukowo-Techniczne, 2003.

[6] Trąbka W., Szpitalne Systemy Informatyczne, Kraków, Uniwersyteckie Wydawnictwo Medyczne "Vesalius", 1999.

[7] Ziniewicz P., Malinowski P., Mnich S. Z., Clinical department information system development, Studies in Logic, Grammar and Rhetoric, 21 (34), 2010.

[8] Ziniewicz P., Milewski R., Malinowski P., Mnich S. Z., Informatyczny system zarządzania jednostką naukowo-kliniczną Uniwersytetu Medycznego, Współczesne wyzwania strukturalne i menadżerskie w ochronie zdrowia, red. R. Lewandowski, R. Walkowiak, Olsztyn: Olsztyńska Wyższa Szkoła Informatyki i Zarządzania im. prof. T. Kotarbińskiego, 2010.