Modelling a Hospital Information System with Decentralized Label Model

Slawomir Holodniuk

Kongens Lyngby 2011
IMM-MSC-2011

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MSC: ISSN 0909-3192

Summary

Electronic Medical Records (EMR) systems are even more popular solutions now-a-days in the health sector. Successful implementation of this type of systems strongly depends on how secure they are. In this master thesis we verify how good can the Decentralized Label Model serve the purpose of implementing a secure EMR system. We also investigate what is the relation between the Decentralized Label Model and Aspect Oriented Programming with respect to meeting security requirements when implementing an EMR system using these two concepts.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling (IMM), the Technical University of Denmark in partial fulfilment of the requirements for acquiring the M.Sc. degree in engineering.

The Decentralized Label Model is a software security framework for providing formal verification of confidentiality and integrity properties of software systems. The thesis examines applicability of the Decentralized Label Model in the area of Electronic Medical Records systems.

The thesis consists of a summary report and a CD with the source code of the implemented system.

Kongens Lyngby, June 2011

Slawomir Holodniuk

Contents

Chapter 1

Introduction

1.1 Electronic Medical Records - motivation and challenges

The Hospital Information System to be modelled is an example of an EMR (Electronic Medical Record) system. The EMR systems aim at replacing (or supporting) the paper-based medical records. The old fashioned style of registering medical information of patients at private General Practitioners offices, hospitals, medical-care companies, and similar tends to be superseded by network-enabled information systems to ease storage, retrieval, sharing and analysis of medical records.

This process started in western countries in 70's - at that point of time the software solutions were used by single departments of hospitals mainly for administrative purposes. As the time went by, these applications were adjusted and extended to support also the work of clinicians. Then, the medical records started being used also in GP's consulting rooms to provide medical information about patients. As the requirement of medical-care support systems shifted from purely administrative to what is understood now as EMR systems, new chances, but also new challenges arose.

At present, the software solutions for medial-records sector seem to be inevitable in a cost-efficient public health care service. The highest-developed countries (as USA, Switzerland and United Kingdom) put great effort and assign huge

amounts of money for pursuing successful nation- or region-wide EMR systems. Public health care service is one of the biggest (with respect to received amount of money) budget beneficiaries. In USA, it consumes yearly around $800 bln, that is 21% of Federal Budget (see figure 1.1). Improvement of the health-care service efficiency may possibly save enormous volume of money. The experience of employing IT in both public and private sectors suggests that using innovative information systems usually effects in major savings.

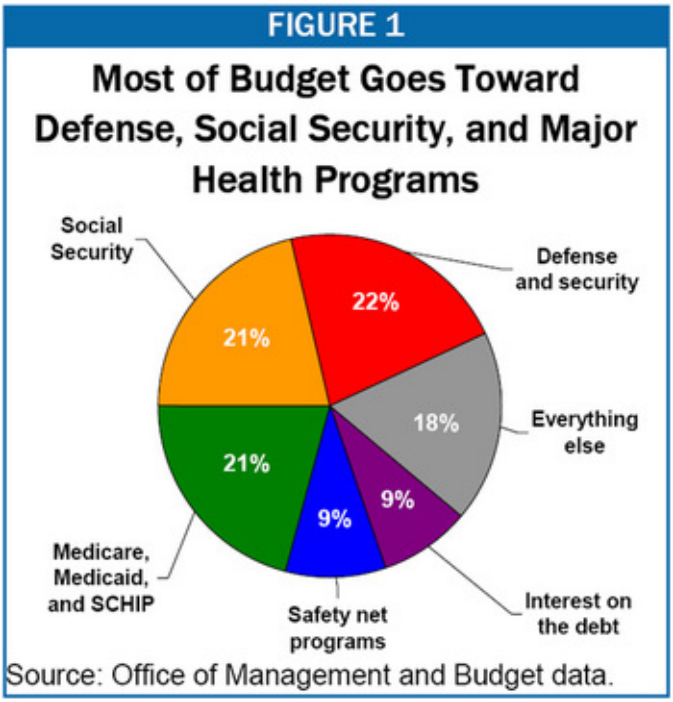

Figure 1.1: Public health service funding from Federal Budget of U.S. [1]

1.2 Decentralized Label Model for the Hospital Information System

The main question this thesis is trying to answer is if the **Decentralized Label Model** is an applicable framework for implementing a secure EMR system.
The Decentralized Label Model is a framework for ensuring confidentiality and integrity of data processed by information systems. It is a very abstract framework - the Decentralized Label Model is only a theoretical concept of how to ensure data privacy. However there exists an implementation of this framework - it is called **Java Information Flow** (JIF). JIF is a **java**-based programming language implementing the features of the Decentralized Label Model. It is developed by i.a. Andrew Myers - a co-author of the Decentralized Label Model. Answering the question of how applicable is the Decentralized Label Model for implementing a secure EMR system, we use the JIF language to implement such a system. The implemented system and the implementation process experiences will be the base for evaluation of the Decentralized Label Model.

Implementation of an EMR system, even when not paying attention to its security properties, is a big task requiring a lot of time and effort. For this reason we will focus mainly on the security properties of the system, and keep the functionality as simple as possible.
The next section presents the security requirements, and justifies them by showing the sources thereof.

1.3 Hospital Information System - gathering security requirements

Development of an information system involves requirements elicitation - a process of establishing requirements for the developed system. A software requirement is a desired property or functionality of a software system. A software system is considered as successful if it meets its requirements. The main source of requirements are the stakeholders - people, groups and organizations that can affect of be affected by a successful or unsuccessful development of the software system. It is only the stakeholders who decide (explicitly or non-explicitly) if the developed system is successful, thus their concerns and wishes about the system are key factors in the software development process.
It is not different in the case of the Hospital Information System. Multiple parties that are involved in the health-care process, law-makers, regulators and

the public opinion do have a stake both in the development process and the operational phase of an EMR the system. To get in this study the modelling of the Hospital Information System as realistic as possible, we will investigate who are stakeholders of a typical Electronic Medical Record system, what are their stakes, and what concerns and wishes for the Hospital Information System they may have. This information is mainly gathered from papers about the challenges in implementing EMR systems [5, 19, 14, 28], specifications of requirements for such system [27, 26, 17], and comparative studies and governmental agencies guidelines [6, 20, 12, 13].

These sources for sure are not exhaustive. First, it seems that a general specification of security requirements for EMR systems has not been yet published. Second, in a real-life software development process (and requirements engineering) an important source of requirements are the people who are to be the users of the system (in this study these are e.g. doctors, nurses and administrative staff of a hospital). Unfortunately, it is infeasible to get in touch with such people for this study. On the other hand, the requirements posed by those people concern mainly the functionality, and the user interface of the system - what features should be there, how the screens should look like, etc. In this study we are concerned mainly with the security requirements, which are not the most important from the medical-care staff's point of view. The mentioned papers seem to be sufficient for the security requirements elicitation process. The main categories of requirements identified for the Hospital Information System are:

- **Data Confidentiality Requirements** - sensitive data protection against unauthorized disclosure
- **Data Integrity Requirements** - sensitive data protection against unauthorized change / deletion
- **Maintainability Requirements** - reconfigurability of the system, easiness of specifying and changing the data security policies
- **Interoperability Requirements** - ability to exchange information with other systems

Requirements falling into these four categories are the most often encountered in the mentioned reference papers, thus it is reasonable to focus on them and put most effort on elaborating them.

Later in this section a few popular and proven requirements elicitation techniques (like stakeholders analysis, goals analysis, business processes analysis

etc.) are used to gather the interesting requirements - those, which are related to security, maintainability and interoperability of the Hospital Information System.

1.3.1 Stakeholders

Project stakeholders are persons, groups and institutions which can affect or can be affected by the project. It is only the stakeholders who decide about a success or a failure of the developed system.
The stakeholders analysis helps us with getting with requirements elicitation closer to a real-life situation, as development of virtually any middle- to large-scale software solution requires recognizing the needs of the project stakeholders. The EMR systems belong to this class of software solutions.

Table 1.1 presents the stakeholders of the Hospital Information System project. Each stakeholder is assigned four attributes:

- **Exposure** - extent to which a success or a failure of the Hospital Information System will affect the stakeholder
- **Power** - the ability of the stakeholder to affect the software development process (also the authority to kill the process) or importance of the opinion, when deciding if the system is a success or a failure.
- **Urgency** - the extent to which the stakeholder can affect the development process immediately
- **Importance** - "average" of the **Exposure**, **Power** and **Urgency**. It indicates how important the stakeholder is.

Each attribute can take one of three values:

- * - low
- ** - medium
- *** - high

Stakeholder	Exposure	Power	Urgency	Importance
Software provider	** (***)	***	***	***
Hospital Managers	***	***	***	***
Medical staff	*	**	*	**
Patients	**	***	*	**
Data Protection Agencies	*	***	*	**

Table 1.1: The stakeholders and their attributes

Software provider

Summary : the software provider is the company that will develop the Hospital Information System. It's exposure is medium to high as selling a system that does not obey the governmental privacy regulations can result in liability which could ruin the company. The software provider may kill the project instantly, thus power and importance are also on high levels.

Stake : the good name of the company and perhaps it's existence

Needs : the software provider looks for a technology that can assure the security requirements of the Hospital Information System will be met. This technology has to have a sound scientific background, as this would dramatically decrease the risk of liability in case of a patient records leak.

Hospital Managers

Summary : the hospital managers are the people who are responsible for running the hospital where the Hospital Information System is to be deployed. They are the customer of the software provider as they make decision about buying the software. The exposure is high as buying and employing a system that does not obey the governmental privacy regulations can result in law liability and a loss of job. On the other hand a successful system can give them a lot of benefits due to savings.

Stake : their job positions

Needs : the hospital managers have to be sure that the system they buy does obey the data privacy regulations. System security and reliability are the top required qualities.

Medical staff

Summary : the medical staff are the target users of the Hospital Information System. They are to operate on the system on a daily basis - it should become their working tool. In case of a successful implementation the medical staff will have the work easier and their working conditions are improved. Otherwise the medical staff is not affected too much, thus the exposure is medium. The medical staff may decide not to use the system if they find it poor and unpleasant, thus power is medium. However this can only happen when the system is already deployed, that is why the urgency is low

Stake : better working conditions

Needs : the medical-care staff wants the Hospital Information System to be user-friendly, fast, responsive and reliable.

Patients

Summary : the patients are the subjects for the Hospital Information System. The Hospital Information System is expected to process the patient records. The exposure of the patients is medium as in the case the system is unsuccessful and their medical data leaks, their privacy is severely damaged and if it is successful they will notice improvement in the service quality. The urgency is low as the patients may only complain about the system after it is deployed and operating.

Stake : the privacy of their data and the quality of medical-care service

Needs : the patients wants the system to be fast (shorter waiting time), reliable and protect their privacy

Data protection agencies

Summary : the data protection agencies are public law enforcement agencies concerned with protection of the citizens' privacy. They are not affected by the system, thus their exposure is low. The urgency is also low as only when the system is operating and a data leak happens they can act. However, in such situation the agencies can order to shut down the system and take appropriate legal steps against the software provider and the hospital managers

Stake : the data privacy of the citizens

Needs : the Hospital Information System should obey regulations concerning patient records privacy

1.3.2 The Hospital Information System goals

Goal analysis helps identifying the goals of the developed system.
The goal analysis in this study is restricted only to the security-related goals. The multitude of non-security goals for such an information system is usually overwhelming, yet not a must to look after as far as the system's security is concerned.

An arrow in the diagram shows the "sub-goal" relationship, whereas a line between two goals (round-tangles) with label *conflict* indicates that two goals are in conflict.
The gray background color of a round-tangle indicates that the goal is of the top priority.

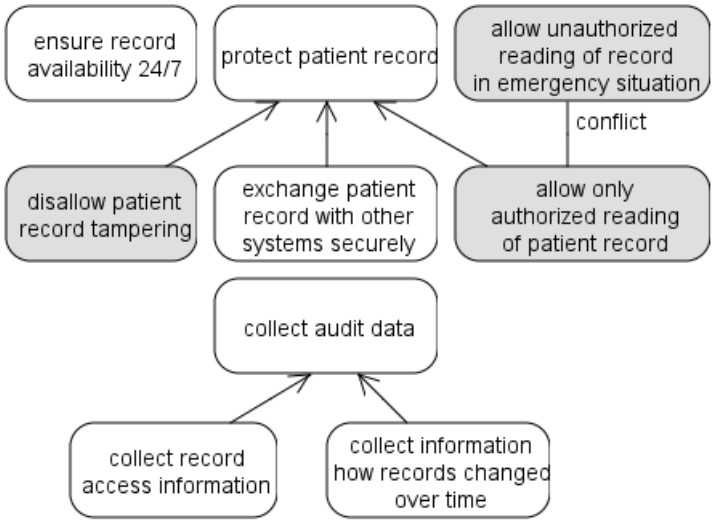

Figure 1.2: The security goals of the Hospital Information System

1.3.3 Business Processes Analysis

Business processes are the operations performed by an organization to meet its goals. Subset of these operations is to be improved and (partially of fully) automatized by information systems. Likewise the Hospital Information System is expected to provide improvement and automation of some business processes of a hospital. As the scope of this project and focus are restricted to security attributes of the Hospital Information System, we will not put effort on the non-security related aspects of the business processes. However, the knowledge and analysis of them is necessary to derive the security requirements, as they shall reveal (at least indirectly) i.a. the information flow in the system, which is a basis for specifying desired confidentiality and integrity properties of the data flowing through the system.

First, we will sketch a simplified medical-care staff hierarchy of an imaginary hospital to identify who are the principals that are expected to use the Hospital Information System.
Second, we will briefly discuss the most common business processes that are expected to involve usage of the Health Information System. From this discussion we will derive the basic security policies following the need-to-know principle - who needs to access which data to perform his tasks.

1.3.3.1 Medical-care staff organizational Structure

For the sake of clearness and abstraction the organizational structure (presented in figure 1.3) is very basic and simplified. It lists the medical-care staff and depicts the subordination relationships. This structure conveys an implicit information: both the *Chief Nurse* and the *Head of Ward* should be granted privileges that are supersets of the union of all subordinates' privileges.

1.3.3.2 Business Processes

The business processes diagram (figure 1.4) and the business processes table (table 1.2) present the most common and important (from the security point of view) business processes of a hospital. Registration, assignment of staff, treatment and making out a patient is a main business scenario for a hospital, therefore the business processes that comprise this scenario are in the area of interest in this analysis.

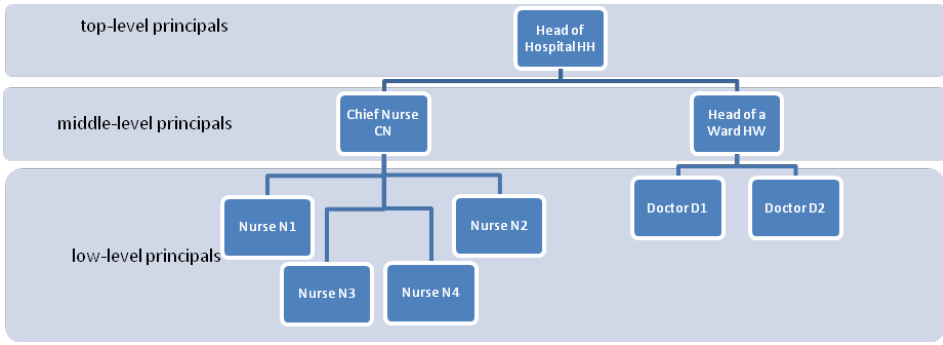

Figure 1.3: The medical-care staff structure of a sample hospital

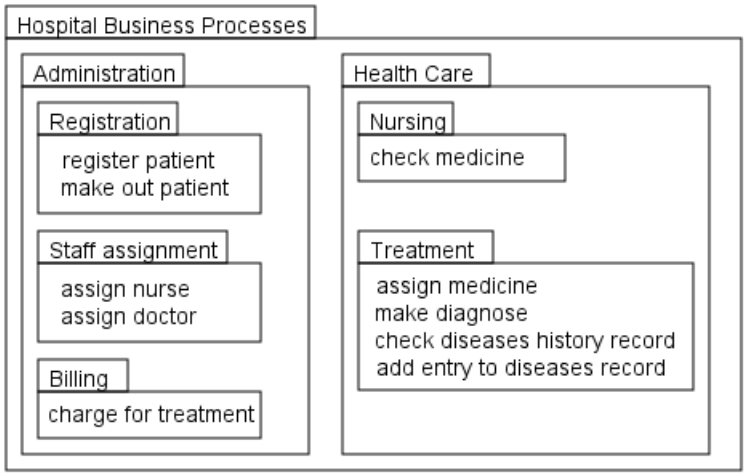

Figure 1.4: Business processes diagram for a sample hospital

1.3.4 Requirements

The mentioned reference papers and the analysis done thus far allow to state following security-related requirements for the Hospital Information System:

R1 : patient record should be accessible (reading and writing) only for persons and parties explicitly named by the patient. Exception: requirement **R7**

R2 : the medical part of a patient record should be accessible only to the

medical-care staff

R3 : the administrative part of a patient record shall be accessible only to the administrative staff

R4 : the medical part of the patient record should be accessible only to the medical-care staff members explicitly assigned to the patient

R5 : the users may only append information to the patient record - no information shall be altered

R6 : the unauthorized attempts to access a patient record shall be logged

R7a : reading a patient record by unauthorized parties shall be allowed in emergency cases.

R7b : event described in **R7a** shall be logged.

R8 : it should take not more than one hour for a trained person to specify and re-configure simple patient record security policies. The training should take no longer than one week.

R9 : enforcement of a newly specified patient record security policy should take no more than one day

R10 : the Hospital Information System shall authenticate the users

R11 : the Hospital Information System shall provide an interface for communication with other EMR systems

R12 : a patient record exchanged with other system should be transmitted securely

R13 : the technology employed for system implementation shall have sound scientific background

R14 : the Hospital Information System shall implement following authentication schemes: username-password and smart-cards.

Domain	Process	Description
Registration		
	register patient	add a new patient to the registries
	make out patient	archive patients record
Staff assignment		
	assign nurse	assign a nurse for a patient
	assign doctor	assign a doctor to a patient
Billing		
	charge for treatment	add a payment information for patient treatment
Nursing		
	check medicine	check what medicine to give to a patient
Treatment		
	assign medicine	assign medicine to a patient
	make diagnose	write down a diagnose for a patient
	check diseases history record	search a patient's record for previous diseases
	add entry do diseases history record	append to the record information about the current disease

Table 1.2: Business Processes of a sample hospital

Chapter 2

Case study: Hospital Information System

In this chapter we discuss the design of the prototype Hospital Information System.

2.1 Conceptual Design

In this section we present the conceptual design of the Hospital Information System.

First we need to mention, that design of the Hospital Information System is not a simple task, as there are two conflicting circumstances related to the prototype system:

- the system should be as realistic as possible in order to correctly validate the Decentralized Label Model in the area of Electronic Medical Recordsystems.
- the system should be kept simple due to implementation time constraints

For these reasons the conceptual design grasps only the most representative data items that can be found in Electronic Medical Recordsystems. The selected data items are the most common for these system and seem to be sufficient for evaluating the privacy protection ensured by Decentralized Label Model, later on. Also the internal complexity of the system is reduced. Normally, large information systems (like EMR systems) involve deployment of the software on many machines (they are distributed) and work together with other external systems. Due to mentioned implementation time constraints we design a single self-containing application deployed at one host.
On the other hand the system is expected to cooperate with databases, which makes it much more realistic, as a vast range of information systems now-a-days retrieves from and uploads processed information to databases.

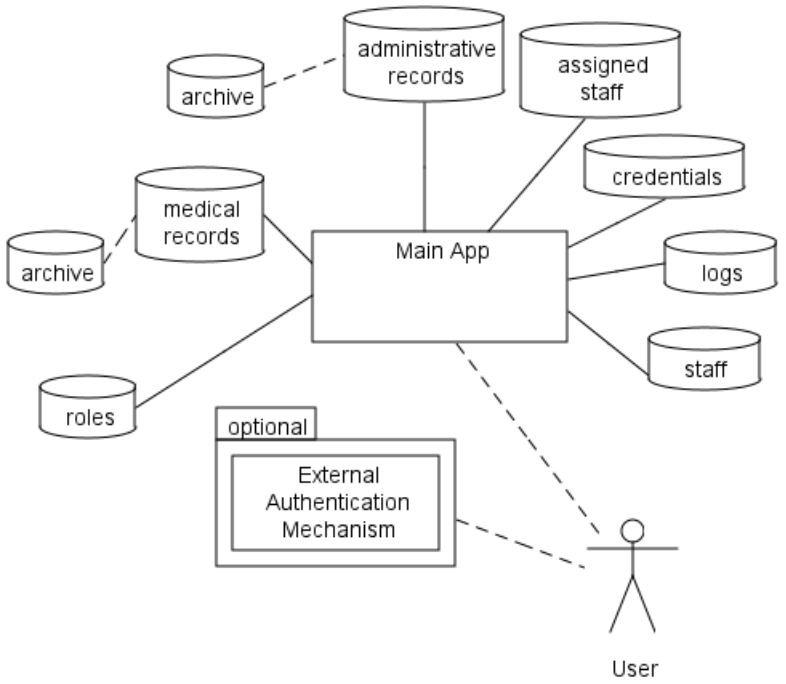

Figure 2.1: The conceptual design of the Hospital Information System

The figure 2.1 presents graphically the conceptual design of the Hospital Information System. The box in the middle (*MainApp*) symbolizes the self-containing system which will process the patient record data. It is surrounded by a number of cylinders representing the databases, these are:

- **administrative record** - here resides all the administrative information about the patients, like their name, next of kin, social security ID etc.

- **assigned staff** - every patient is assigned a doctor and a nurse taking care of him. These staff members have got special privileges for accessing the record of the patient. This database contains information about assignment of the medical staff to the patients.

- **credentials** - the users have to log in to the system before they can perform any action on it. In this table the $< user-name, hashedpassword >$ pairs are stored. The authentication of the users is done against this database.

- **logs** - the system is expected to log the security-related events. The logs go into this database.

- **medical records** - here all the health-related information about the patients is stored, like what treatments they are undergoing, what medicine they are taking, what allergies they have got, etc.

- **roles** - the staff members of a hospital using the Hospital Information System assume various roles, like doctor, nurse or an administration clerk. This database embraces the information what roles the staff members assume.

- **staff** - here the information about the hospital staff is stored

For finer grained description of the databases design please refer to table 2.7.

Below the *MainApp* box there is another box symbolizing an optional authentication mechanism - this external mechanism could be used for authentication of the users if the password scheme is not sufficient. However authorization of the action performed by users is expected to done by the *MainApp*.

Summarizing the conceptual design, there is one self-containing application cooperating with a number of databases and possibly an external authentication mechanism. Now we will go to the functional design of the system, presented in the form of use cases.

2.2 Use cases

In this section we display the functional design of the Hospital Information System.

The required functionality of the system is presented in the form of use-cases: simple usage scenarios. There are following use-cases identified for the system:

- **authenticate** - authenticate a user to the system
- **create user** - add a new user (hospital staff member) to the system
- **read medical record** - get the medical information from the patient record
- **update medical record** - update the medical information in the patient record
- **read administrative record** - get the administrative information from the patient record
- **update administrative record** - update the administrative information in the patient record
- **assign staff** - assign a doctor or a nurse to a patient

The overview of the use cases is presented in the use case diagram (figure 2.2). The detailed description of the use cases is presented in tables 2.1, 2.2, 2.3, 2.4 ,2.5 and 2.6.

The sequence diagrams 2.3, 2.4, 2.5, 2.6 and 2.7 present the control-flow in the use cases. There is no sequence diagram for the use case *authenticate* as the control flow in this use case depends on the chosen *external authentication mechanism* (if decided to use it).

2.3 Databases design

Table 2.7 presents the design of the above mentioned databases the *MainApp* is interacting with.

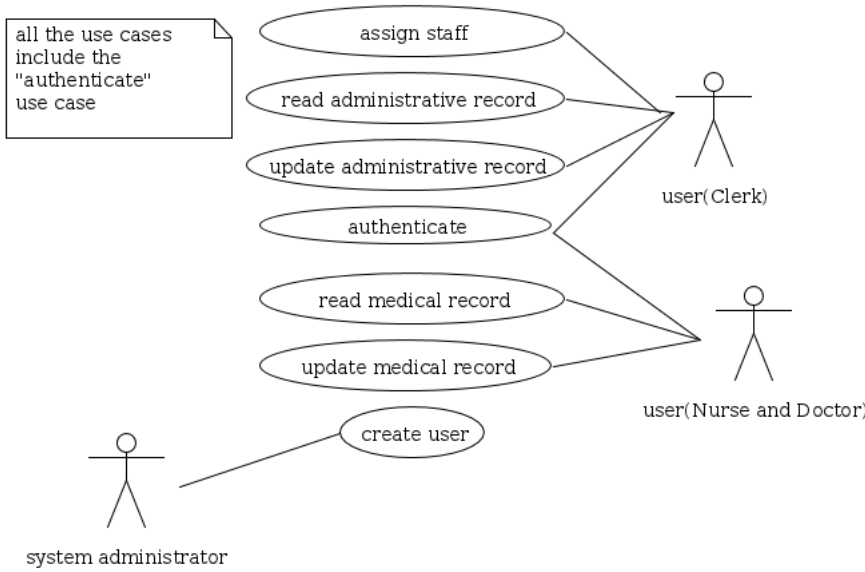

Figure 2.2: The use cases of the Hospital Information System

Use case name:	authenticate
Summary:	checking the identity of a user
Actors:	a user U
Pre-conditions:	-
Basic scenario:	1. U types-in his user-name 2. U types-in his password
Alternative scenarios:	an external authentication mechanism can be used

Post-conditions:	if the credentials were correct:
	- U is authenticated and the system displays command prompt
	- the log-in event is logged
	else:
	- U is informed about incorrect credentials

Comments:	-

Table 2.1: *authenticate* use-case

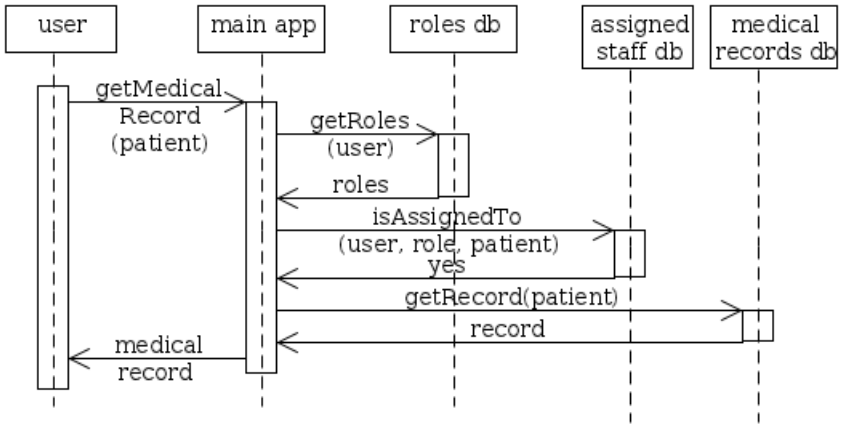

Figure 2.3: Sequence diagram of the reading medical record use case

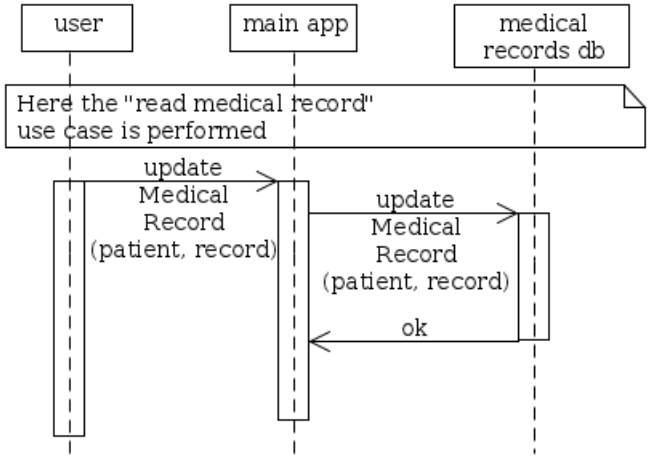

Figure 2.4: Sequence diagram of the update medical record use case

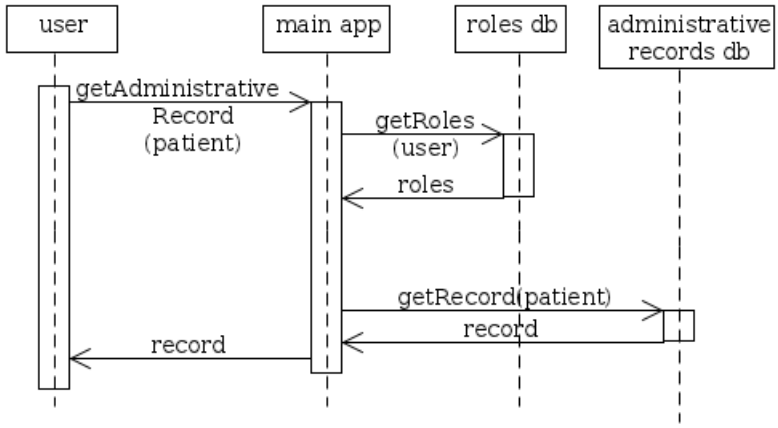

Figure 2.5: Sequence diagram of the reading administrative record use case

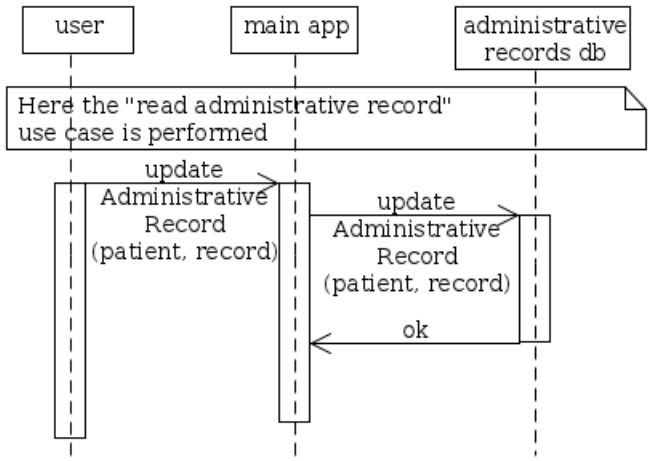

Figure 2.6: Sequence diagram of the update administrative record use case

Use case name:	read medical record
Summary:	a user reads the medical record of a patient
Actors:	a user U
Pre-conditions:	U is authenticated
Basic scenario:	1. U performs operation of reading the medical record of a patient
Alternative scenarios:	-
Post-conditions:	if U is a medical staff member to authorized to read data from the medical record - the medical record is displayed to U else: - the event of issuing a unauthorized operation is logged
Comments:	-

Table 2.2: *read medical record* use-case

Use case name:	update medical record
Summary:	a user updates the medical record of a patient
Actors:	a user U
Pre-conditions:	U is authenticated
Basic scenario:	1. U performs operation of updating the medical record of a patient
Alternative scenarios:	-
Post-conditions:	if U is a medical staff member to authorized to write data of the medical record - the medical record is updated else: - the event of issuing a unauthorized operation is logged
Comments:	-

Table 2.3: *update medical record* use-case

Use case name:	read administrative record
Summary:	a user reads the administrative record of a patient
Actors:	a user U
Pre-conditions:	U is authenticated
Basic scenario:	1. U performs operation of reading the administrative record of a patient
Alternative scenarios:	-
Post-conditions:	if U is a *Clerk* - the administrative record is displayed to U else: - the event of issuing a unauthorized operation is logged
Comments:	-

Table 2.4: *read administrative record* use-case

Use case name:	update administrative record
Summary:	a user updates the administrative record of a patient
Actors:	a user U
Pre-conditions:	U is authenticated
Basic scenario:	1. U performs operation of updating the administrative record of a patient
Alternative scenarios:	-
Post-conditions:	if U is a *Clerk* - the administrative record is updated else: - the event of issuing a unauthorized operation is logged
Comments:	-

Table 2.5: *update administrative record* use-case

Use case name:	Assign Staff
Summary:	a user assigns medical-care staff to a patient
Actors:	a user U
Pre-conditions:	U is authenticated
Basic scenario:	U assigns a medical-care staff member to a patient
Alternative scenarios:	-
Post-conditions:	if U is a *Clerk* - the medical-care staff member is assigned to the patient else: - the event of issuing a unauthorized operation is logged
Comments:	-

Table 2.6: *assign staff* use-case

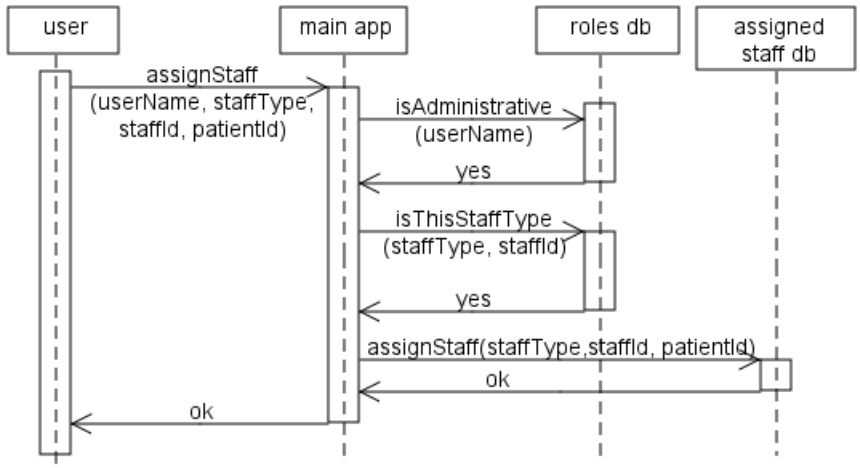

Figure 2.7: Sequence diagram of the staff assignment use case

Database	Entity	Description
Roles		the roles of staff (doctor, nurse etc.)
	staffId	unique identifier of staff member
	role	a role the staff member may assume
Staff		the hospital staff
	staffId	unique identifier of staff member
	userName	system user-name
	personalData	-
	qualifications	professional qualifications
Assigned Staff		assignment of doctors and nurses to patients
	patientId	unique identifier of a patient
	doctor	unique identifier of a staff member (doctor)
	nurse	unique identifier of a staff member (nurse)
Credentials		credentials of system users
	userName	system user-name
	hashed password	-
	personalData	-
	expires	date when the account expires
Administrative Record		administrative part of patient records
	patientId	unique identifier of patient
	personalData	-
	ssID	social security identifier
	nextOfKin	contact to a relative of the patient
Medical Records		medical part of patient records
	patientId	unique identifier of patient
	diseasesHistory	a list of diseases the patient went through
	currentMedicine	the medicine taken currently by the patient
	allergies	patient's allergies for food, medicine etc.
	specialNote	doctor's special note about the patient
	treatmentHistory	the measures taken to treat the patient

Table 2.7: Design of HIS databases

Chapter 3

Decentralized Label Model

In this chapter we explain what is the Decentralized Label Model. This chapter is based on the articles presenting the model ([22, 24, 23]) and the on-line JIF manual [2].

3.1 Access control

The requirements for the Hospital Information System stated in 1.3.4 related to the data confidentiality and integrity (**R1-5,7**) demand implementation of an access control mechanism. Meeting these requirements is only possible if the system can authenticate the users and authorize their actions on the patient records. These two functionalities(authentication and authorization) comprise access control - an extremely extensively researched topic in the computer security.
Authentication is a process of verifying individual's identity. This step precedes authorization. One needs to prove identity in order to be recognized by a system as a legitimate user.
Authorization is a process of deciding if the authenticated user attempting to perform an action is allowed to do so. In the case of the Hospital Information System the subject of the authorization process are the hospital staff members using the Hospital Information System. They need to prove their identity before

performing any actions in the system. After authentication the hospital staff members can do actions on the system, e.g. read a medical record of a patient. This action however is restricted to the medical staff only, so the authorization mechanism should check if the user attempting to perform the reading medical record action is a member of the medical staff.

Now we will present the Decentralized Label Model and explain how it could provide access control for the data processed by the Hospital Information System. Explaining the Decentralized Label Model will be based on the case study of the Hospital Information System, as this will trim the extent to which the explanations reach and should help understanding the the case study itself.

3.2 Basics

The Decentralized Label Model is a framework for ensuring data confidentiality and integrity in software systems. The basic concept behind this framework is following: every data entity (e.g. a database record, a text read from console etc.) in a system is associated with security policies for this entity. These security policies tell who is allowed to read from and write to the respective data entity. A set of policies for a data entity is called a **label**.

The figure 3.1 presents a part of the Hospital Information System from the perspective of the Decentralized Label Model. It shows schematically what are the policies like in the Hospital Information System. In the middle we have got the *MainApp* - the self-contained application processing patient records. It reads and writes the administrative record from the "Administrative Records" database. Data transferred between these two are symbolised by a rectangle called "Administrative Record". Underneath the name of the rectangle there is a string $\{Patient \rightarrow Clerk; Patient \leftarrow Clerk\}$. This is the label of this data entity. The first security policy ($Patient \rightarrow Clerk$) says that the owner of the policy is the *Patient* and he allows the *Clerk* to read this data. The second security policy ($Patient \leftarrow Clerk$) says that the owner of the policy is the *Patient* and he allows the *Clerk* to write this data. The security policies are separated by a semicolon (';').
Similar situation is with the **interface** data. The box called "interface" symbolises the information displayed to the user and the input of the user. The label for this data entity is $\{\star \rightarrow User; \star \leftarrow User\}$. It says that the owner of the security policies is top principal (denoted as '$\star$') and that the *User* can read from and write to this data entity. More thorough explanation of the security

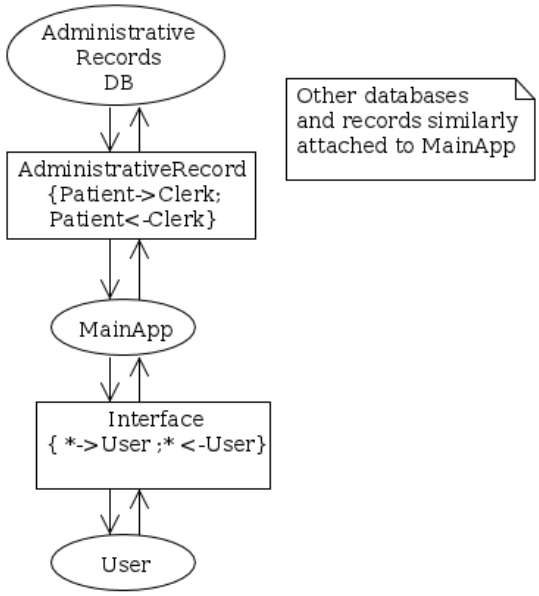

Figure 3.1: The Hospital Information System from the Decentralized Label Model perspective

policies and principals is given in later on in this chapter.
The figure 3.1 presents only a small segment of the whole system. There are many more databases and information entities with various labels, but they are organised in the similar manner as the *Administrative Record*.

3.2.1 Security policies

The security policies comprising labels are of two types:

- confidentiality policies: one principal grants another principal the right to read a data entity
- integrity policies: one principal grants another principal the right to write to a data entity

The notation of the security policies is following:

- **confidentiality policy:** $o \rightarrow r$ - principal o allows principal r to read
- **integrity policy:** $o \leftarrow w$ - principal o allows principal w to write

3.3 Labels

The security policies combined using **join** ($\sqcup$ or ';') and **meet** ($\sqcap$) operators form labels. A label consist of confidentiality policies and integrity policies, which are mutually independent. That means, the meet and join operators work only on pairs of policies of the same kind: pairs of confidentiality policies and pairs of integrity policies The notation used for the labels is following:

- $C(l)$ - confidentiality policies of label l
- $I(l)$ - integrity policies of label l

In the previous section we have seen examples of labels, one of which was $l_1 = \{Hospital \rightarrow Clerk; Hospital \leftarrow Clerk\}$. The above functions applied to this label are:

- $C(l_1) = \{Patient \rightarrow Clerk\}$ - confidentiality policies of label l_1
- $I(l_1) = \{Patient \leftarrow Clerk\}$ - integrity policies of label l_1

3.4 Labels ordering

The labels in the Decentralized Label Model form a lattice. Lattice is a partially ordered set (**POSET**) in which for any pair of elements (e_1, e_2)there exists a supremum (lowest upper bound (LUB) of these two - $LUB(e_1, e_2)$, also known as their *join* - $e_1 \sqcup e_2$) and infimum (greatest lower bound (GLB) of these two - $GLB(e_1, e_2)$, also known as their *meet* - $e_1 \sqcap e_2$). The ordering of labels in the Decentralized Label Model is specified as follows:

Ordering of labels

for labels l_1 and l_2:

$l_1 \sqsubseteq l_2 \iff C(l_1) \sqsubseteq_C C(l_2) \wedge I(l_1) \sqsubseteq_I I(l_2)$

which we read:

"l_2 is at least as restrictive as l_1 if and only if the confidentiality policy of the l_2 is at least as restrictive as the confidentiality policy of l_1 and the integrity policy of the l_2 is at least as restrictive as the integrity policy of l_1".

To define ordering of confidentiality policies a function *readers* is introduced:

Readers function

$readers : P \times C \to 2^P$
$(p, o \to r) \mapsto \{q | o \succeq p \Rightarrow q \succeq o \vee q \succeq r\}$

where P is the set of all principals, C is a set of all possible confidentiality policies over P , and p, o and r are principals.

The function returns for a given principal p and a confidentiality policy $(o \to r)$ a set of principals that p permits to read the data entity.

The ordering of confidentiality policies is defined as follows:

Confidentiality policies ordering

for two confidentiality policies c_1 and c_2:

$c_1 \sqsubseteq_C c_2 \iff (\forall p \in P)\ readers(p, c_1) \supseteq readers(p, c_2)$

which we read:

"confidentiality policy c_2 is as restrictive as confidentiality policy c_1 if and only if for all principals the readers set of c_1 is a superset of the readers set of c_2"

In a similar (actually dual) way is defined the relation between integrity policies. A function $writers$ is introduced:

Writers function

$writers : P \times C \to 2^P$
$(p, o \leftarrow w) \mapsto \{q | o \succeq p \Rightarrow q \succeq o \vee q \succeq w\}$

where P is the set of all principals, C is a set of all possible integrity policies over P , and p, o and r are principals.

The function returns for a given principal p and an integrity policy $(o \leftarrow r)$ a set of principals that p permits to write to the data entity.
The ordering of integrity policies is defined as follows:

Integrity policies ordering

for two integrity policies i_1 and i_2:

$i_1 \sqsubseteq_I i_2 \iff (\forall p \in P)\ writers(p, i_1) \subseteq writers(p, i_2)$

which we read:

"integrity policy i_2 is as restrictive as integrity policy i_1 if and only if for all principals the writers set of i_1 is a subset of the readers set of i_2"

3.5 Label checking

The fundamental and key idea in the Decentralized Label Model is the way, how the security policies are enforced. In order to prove, that the security policies specified for the data entities within the system hold, the information flow in the system is analysed. A special program (called **verifier** , which in fact is a compiler) analyses the information flow in the system and based on that it can prove, that the security policies are not violated.
The security policy determines the security classification (the label) of the associated data entity. Knowing the information flow in the system, it is possible to say if a data entity of higher security classification is written into a data entity of lower security classification. Should this happen, the verifier will tell that there is a possible read up / write down (c.f. *Bell-LaPadula* model) occurring, and this way a security policy violation is reported.
The data entities security classifications (labels) are partially ordered. Hence, it possible to say that one label is more **restrictive** than another one, unless the labels are incomparable. In such cases the Decentralized Label Model is conservative - if there is a data flow between data entities which labels are comparable, a policy violation is reported.

Chapter 4

Case study: realisation in JIF

In this section we present how was the Hospital Information System implemented in the Java Information Flow language [21], the challenges and problems during the system implementation process, and restrictions of the JIF encountered during the implementation.
To see the detailed technical specification of the implementation please refer to the appendix A.1.

4.1 General

The JIF implementation of the Hospital Information System follows the design specified in the chapter 3. The core software application called in the design section *MainApp* has been implemented as a JIF program consisting of three logically separated packages:

- *record* - the classes in this package are a "mapping" of the patient data stored in the databases. These classes are responsible for fetching the actual data from the databases, storing them while the program is running, and uploading the modified data to the databases. They provide interface

for the patient record data modification. The package name is *record*, as the classes in it cater processing of the data that comprise the patient record - both medical and administrative information.

- *interface* - this package embraces all the classes that constitute the user interface (or rather *View* component of the *Model-View-Controller* design pattern) and software interfaces (communication with SQL databases). They facilitate communication between the *MainApp* and the user, and between *MainApp* and the databases.
- *utils* - this package accommodates classes that provide features of logging, authentication and some other neat functionalities used throughout the system

4.2 Data labels design

Table 4.1 presents the labels of the data processed by the Hospital Information System . These labels contain the security policies concerning the data processed by the system.

The columns of the table are:

- **table** - the name of the table from which the data is pulled out
- **column** - the name of the column in the table where the data is located
- **label** - the label the data is labelled with

The databases themselves are not labelled, as there is now way to do it in JIF. Instead, the data pulled out from these databases is labelled and as labelled types starts circulating in the program.

4.3 Meeting the security-related requirements

In the section 1.3.4 we discussed the security requirements for the Hospital Information System. Now we specify the labels for for the data entities circulating

Table	Column	Label
Roles Staff Assigned Staff Credentials	*all*	$\{Clerk \rightarrow *;$ $Clerk \leftarrow *\}$
Administrative Record[patientId]	*all*	$\{Patient[patientId] \rightarrow Clerk;$ $Patient[patientId] \leftarrow Clerk\}$
Medical Record[patientId]	*all*	$\{Patient[patientId] \rightarrow doctorOf(patientId),$ $nurseOf(patientId); Patient[patientId]$
Diseases[patientId] Treatements[patientId]	*all*	$\{Patient[patientId] \rightarrow doctorOf(patientId);$ $Patient[patientId] \leftarrow doctorOf(patientId)\}$

Table 4.1: Design of data labels

in the system. These labels reflected the textual policies conveyed by the security requirements.

4.3.1 Data privacy

The implementation of the system in the JIF language could have been done possibly in at least two general ways:

- *on-the-fly*: the labels would be implemented at the interface between the *MainApp* and the *databases* (see table 2.7) - whenever a query is executed on the database, the query statement and the result of the query execution would be labelled. The labels of the data would be read from one place holding all of them, and put on the program variables "on-the-fly"

- *mapping* - the structure of the data in the database is mapped into Java classes. The classes contain labelled fields - the security policies are placed in the mapping classes, they are sparse.

The first solution (*on-the-fly*) is very attractive, as the labels for all the sensitive data are stored in one place.This should increase maintainability and correctness of the implementation. However implementation of a method for generating labels depending on the data submitted or read from the database is in general not possible - why, it is explained later in this section. Also analysis power of the JIF compiler would be significantly restricted, as the label of the composition of many labelled types it the (least) upper bound of all the components. This way of implementation would imply handling labels that are too restrictive, and in consequence, of no practical use.
The second solution is less convenient - the security policies are sparse, and there is a need to a implement a number of additional Java classes, to map the data from the database. Implementation and maintenance of the security policies scattered in many places in the code is not an easy task. It also increases probability of errors induction. However, this is the feasible way to implement the designed Hospital Information System. And that is why it has been chosen and followed.

Both of these solutions require "hard-coding" of the security policies in the JIF classes in one way or another. Creating labels in the run-time from input streams (text files, databases) is impossible.
The method of static analysis of the code naturally rules this idea out - it is not possible to make a fully conservative static analysis of a program data flow when there is some undetermined input influencing the data. This also applies to the static analysis done by the JIF compiler.
Dynamic analysis of the data flow possibly could cater the feature of generating labels in the run-time, however in the current JIF version (3.3.1) it does not.

4.3.2 Authentication

The requirement **R10** states that the users of the Hospital Information System should be authenticated. Whenever a user tries to log in to the system, he is supposed to proof that he is the one who he claims to be. In the current implementation, the authentication is done on username-password scheme. When the program is started the user is required to input the username and respective password. The credentials are stored in the database in a form of <username,hashed-password> pair. The hashing function is *SHA-1*.

4.4 Package *record*

Having discussed the general matters concerning the JIF implementation of the Hospital Information System, no we will go through and discuss the class diagrams of the three packages comprising the Hospital Information System system:

- package *record*
- package *interface*
- package *utils*

Figure 4.1 presents all the classes comprising the *record* package. These are:

- *MedicalRecord* - a class mapping a number of general medical data, like allergies or currently taken medicine.
- *Diagnoses* - a class mapping medical diagnoses (statements describing what diseases have been detected when a doctor examined the patient).
- *Treatments* - a class mapping information about what treatments the patient has undergone
- *AdministrativeRecord* - a class mapping administrative-related information about a patient, e.g. name, next of kin etc.
- *AssignedStaff* - a class mapping data about designation of a doctor and a nurse to a patient
- PatientsList - a class mapping the list of all patients enrolled in the system

The reason behind mapping the data from the database into JIF classes is very important and reflect the idea for ensuring data confidentiality and integrity in the Hospital Information System. And the reason is that the labelling of the data (in Decentralized Label Model sense) is done in the mapping classes. The fields in these classes are of type *labelled-type* - a JIF type including both the data type (e.g. *String*, *int* etc.) and label representing the policy related to field variable. For example the field *currentMedicine* is a labelled-type:

($String$,$\{Patient \rightarrow Doctor, Nurse; Patient \leftarrow Doctor\}$] meaning that the data type is *String* and the security policy (*label*) is that the *Patient* allows to read this information by the *Doctor* and the *Nurse*, but writing this information is restricted only to the *Doctor*.

Almost all the classes in this package (package *record*) implement methods for loading the data from the database (*load*), uploading them into the database (*upload*), and creating a new entry in the database (*create*). This state of matters also flows out of the decisions to implement the data security in the mapping classes.

To give an impression how the mapper classes are like, we will briefly discuss a part of such a class , namely *Diagnoses* (listing 4.4). This class maps the diagnoses made to the patient by his doctors.
At the beginning of the listing there is a declaration of a label called *dataLabel*. This is the label used for diagnoses data. It says that the *Doctor* can read and write this data. The policy saying that the *Sysroot* allows itself to write the data is there solely for implementation reasons - but we can forget it as this policy only raises the security classification of the label.
The label of the label (as labels are also labelled types in JIF) is $\{\star \leftarrow \star\}$ - the safest (with respect to integrity) label - nobody can write to a variable with this label. This way we ensure that the labels are not overwritten, which could cause a security breach. All the labels in the implementation of the Hospital Information System follow this scheme.
Later on there is declaration of a table holding the diagnoses. It is labelled with label *dataLabel*.
Then we have got a method of adding a diagnose (called *addDiagnose*) accepting following parameters:

- *currentUser* - the role the user is playing in the system
- *currentUserId* - the id of the user (*staffId*)
- *diagnose* - the diagnose to be added

Virtually any method in the system accepts the first two parameters - they are necessary for the *caller* clause (*currentUser*) and for logging (currentUserId). The *caller* clause says with whose authority the method needs to be invoked. This mechanism assures that the caller principal (*currentUser*) is the real caller. First statement in the method is an **act for** statement - it checks if the principal current user is granted all the privileges of a doctor (in fact if he is a doctor). If

the condition is true, a new diagnose is added to the database. Else the attempt of illegitimate operation is logged. In the former case, the SQL command is created from variables. What is worth noticing in here, is the parametrization of the *SQLQueriesHandler* - the parameter is the label *dataLabel* - the same the diagnoses are labelled with. This assures that only variables of less or equal restrictiveness can comprise the SQL command - this way data leakage is prevented on the point of interaction between the JIF program and the database.

```
// class for handling diagnoses of made for a patient
class Diagnoses {

    // with this label the diagnoses should be labelled
    final label {*<-*} dataLabel =
        new label{Patient->Doctor; Patient<-Doctor; Sysroot<-*};

    // chronological list of diagnoses
    public String{*dataLabel}[]{*dataLabel} diagnoses = null;
    ...

    public void addDiagnose Sysroot<-*}
        (principal{Sysroot<-*}currentUser,
        String{Sysroot<-*} currentUserId,
        String{Sysroot<-*;*<-currentUser} diagnose)
        where caller(currentUser){

        if(currentUser actsfor Doctor){
            try{
                SQLQueriesHandler[{*dataLabel}] handler =
                    new SQLQueriesHandler[{*dataLabel}]();

                handler.executeUpdate(
                    "INSERT INTO Diagnoses (patientId," +
                    "diagnose, author, date) VALUES (" +
                    "'" + patientId + "'" + ",'" +
                    diagnose + "'" + ",'" + currentUserId +
                    "'"+ "," + "NOW()"+ ")"
                );
            }catch(Exception ex){}
        }else{
            Logger.log(currentUser,"unauthorized command",
                "user tried to add a diagnose to the patient: "
                + patientId);
        }
    }
    ...
}
```

Listing 4.1: Part of the class *Diagnoses*

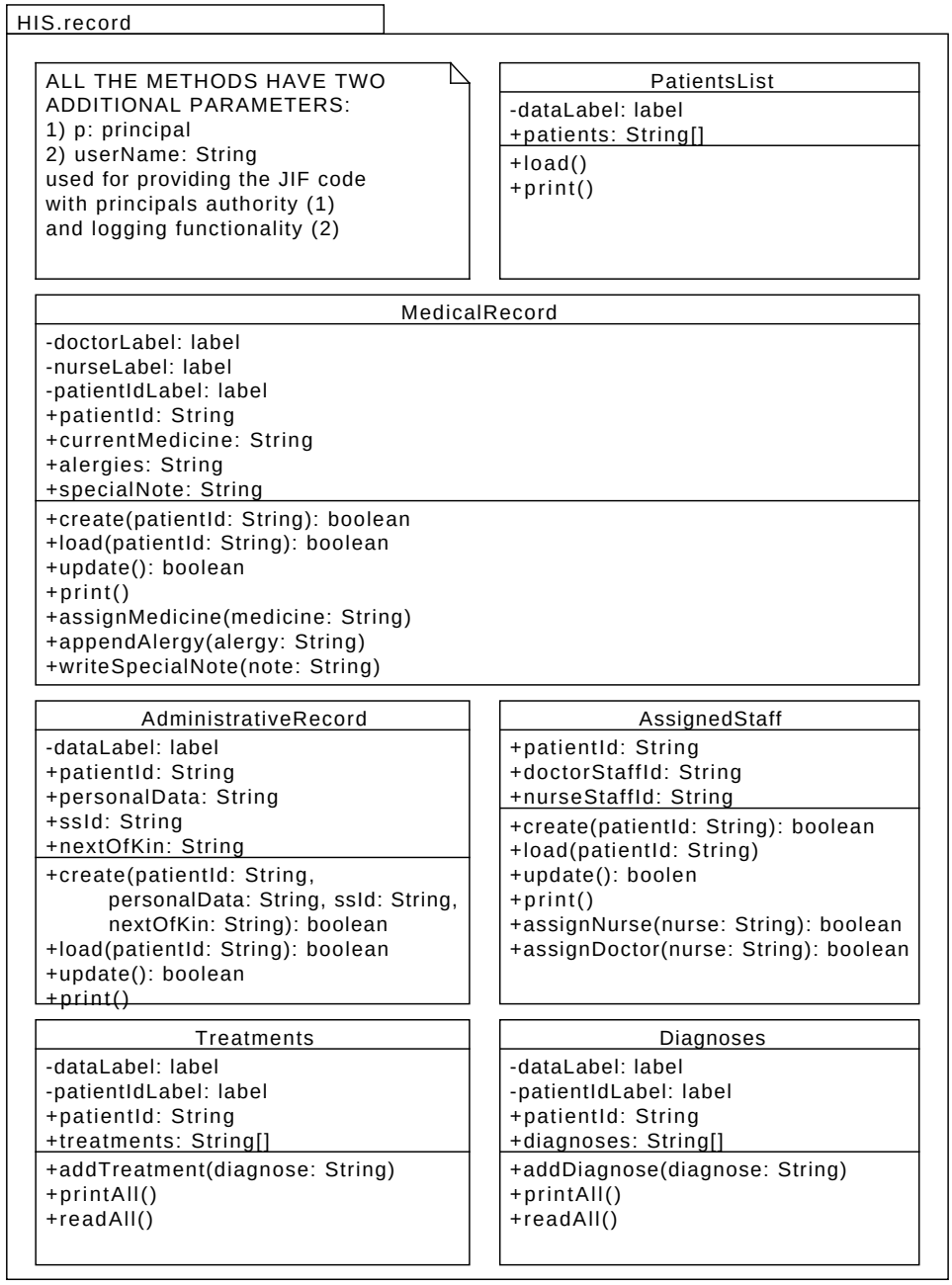

Figure 4.1: The UML class diagram of the *record* package

4.5 Package *interface*

The package *interface* gathers together classes that facilitate the communication between:

- user and *MainApp* (let us denote this interface *U2App*) - the user of the Hospital Information System interacts with the system by a command prompt.
- the *MainApp* and the SQL database (let us denote this interface *App2Db* - the data processed by the *MainApp* are permanently stored in a database. Therefore there is a need to communicate this data.

These interfaces are a not Java interfaces - they are just collections of classes facilitating communication between the system actors.

The interface *U2App* consist of three classes:

- *Main* - the main class of the system. The method *main* of this class is the starting point of the program execution. This method reads from the input parameters the *id* of the patient for whom the program is started. Only knowing that the program can tell what is the role of the user (e.g. *Doctor*, *Nurse*, etc.) for the specified patient
- *Console* - this class allows the *MainApp* to read from and write to the standard output. The standard Java streams *System.in* and *System.out* are blocked in JIF - they cannot be used, perhaps for security reasons: they are not labelled-types and therefore cannot be assigned a label. This prevents establishing the sensitivity of the data communicated through these streams.
- *CommandsProcessor* - here the commands typed-in by the user are parsed, validated and eventually executed. For each command there is a private method that executes a valid command, e.g. by calling the method *AssignedStaff.assignDoctor* when "assignDoctor" command was issued by the user.

The interface *U2App* is text-based, as the current JIF implementation does not support the Java threaded model. It means, that a JIF program is not able to create threads, which is requisite to implement a **Graphical User Interface**

(**GUI**). In fact, implementing a text-based interface is more time and effort consuming than creating a GUI - modern **IDE**'s offer an easy way to "click-out" a GUI in a few minutes.

As mentioned earlier in this section, the standard Java streams are blocked in JIF. The JIF class *jif.runtime.Runtime* provides methods to access the standard input and standard output. However they are not operational - they are buggy and apparently not fully implemented.
For this reason there was a need to implement the standard input and output operations in some other way. The thing seemed hopeless, but it turned out that there is a way to use plain Java classes (and libraries, for that matter) by

1. creating a JIF "signature" class - one containing only signatures of fields and methods,

2. creating a Java class implementing the methods which signatures are in the "signature" class

This way one can use many handy Java classes that are not implemented in JIF. And this is the way the *Console* class was implemented.

Few more words about the *Main.main* method and the patient id passed as its parameter (see listing 4.5).
Why a run of the program can be done for only one patient at a time? This only restricts the usability of the program, as one needs to re-run the program for different patients.

The answer is: security.

Or more precisely, the poorness of the JIF implementation. The *principal* type in the current JIF implementation is not flexible enough. It is not parametrized - a *principal Doctor* denounce the role of a doctor. It is not possible to distinguish between doctors, and the requirements to the Hospital Information System impose that: doctors are assigned to patients. Say a user A is the doctor of a patient X and a user B is the doctor of the patient Y. Then if A and B are not the same, A is not playing the role of a doctor for Y, and B is not playing the role of a doctor for X either. The role of a user depends on the assignment to patients (see *AssignedStaff* class).

For this reason the patient id must be specified at the beginning of the program, to set up the role of the user.

```
...

class Main authority(Sysroot){

    ...
    public static final void main{*<-*}
        (String{*<-*}[]{*<-*} args)
        where authority(Sysroot){
        ...
        patientId=args[0];
        ...
        String userName=
            AuthenticationManager.authenticate();

        // create the principal depending on the
        // "role" of user
        final principal currentUser =
            AuthenticationManager.role(userName,patientId);
        ...
        /*  Sysroot can act for anybody but this
            check is necessary for static analysis
            during the compliation time*/

        if(Sysroot actsfor currentUser){
            CommandsProcessor cp =
                new CommandsProcessor();
            cp.startSession(currentUser,
                userName, patientId);
        ...
        }
    }
}
```

Listing 4.2: Part of the class *Main*

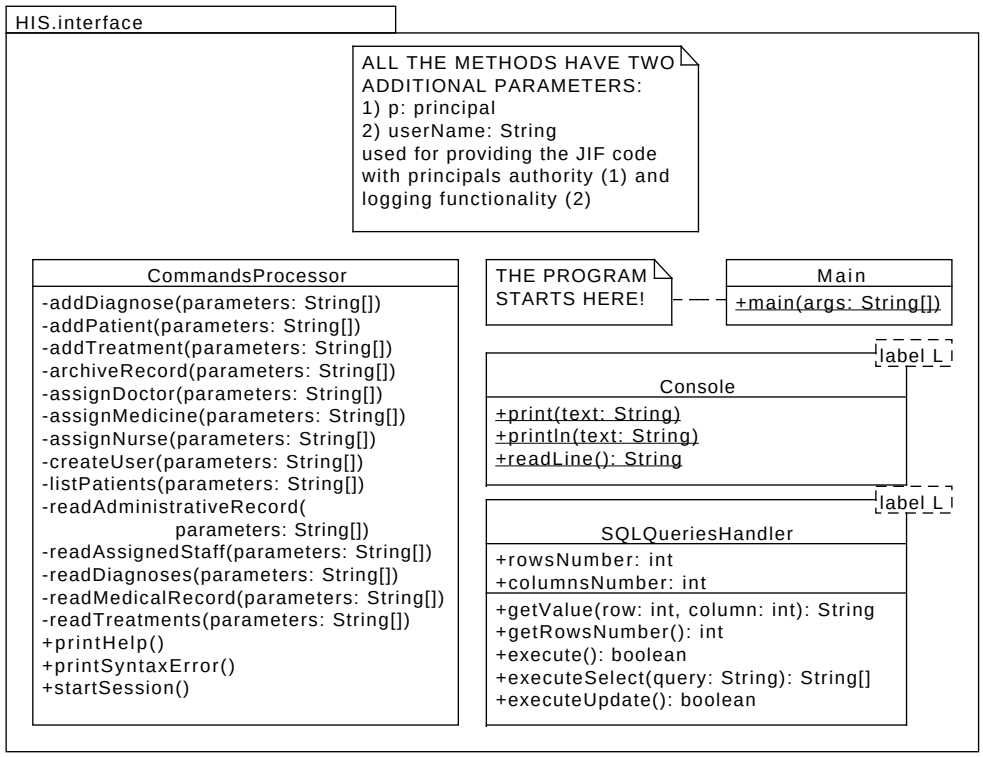

Figure 4.2: The UML class diagram of the *interface* package

4.6 Package *utils*

The *utils* package contains classes which provide functionalities that are not directly related to processing of the patient record data, but are required by the system, like:

- **logging** - inserting into the database log entries when security policies violation attempts happen, e.g. when a *Doctor* attempts to read the administrative data of his patient (this is disallowed)
- **authentication** - verification of the user identity
- ***string* processing** - *string* parsing etc.

The logging class offers a method for inserting a log entry to the database. It takes as parameters the event type (e.g. "unauthorized reading attempt") and a more precise description (e.g. "user *doc1* tried to read the administrative record of patient *p2*"). The logging class is presented in listing 4.6

The authentication is password-based. The hashed passwords are stored in the *Credentials* table. The listing 4.6 presents a part of the *AuthenicationManager* class.
The method *role* returns the role a user is playing in the system (e.g. *Doctor*). The first parameter (*userName*) is the id of the user, the second is the id of the patient (*patientId*) from whose perspective the role is played. For the same user and various patients the roles may be (and usually is) different - a user can be a doctor for one patient, but can be "nobody" (in the roles terms) to another one. It depends on the medical staff assignment to the patient.
The method *authenticate* performs the authentication process - it verifies the credentials submitted by the user. One of the worth noticing points in this method is the declassification of the *userName* variable. The *declassify* construct allows to downgrade the security classification of data. It is requisite at this point of the program, as due to high classification of the password, the *userName* variable is also considered highly confidential (there is a *if* condition on the password variable with the *userName* inside) and needs to be downgraded, so everybody can read it.

```
// Class providing logging functionality for the HIS
class Logger{

    public static boolean log{-<-}
        (principal{-<-} currentUser,
        String {-<-}type, String {-<-} description)
        where caller(currentUser){

        SQLQueriesHandler[{*->*;-<-}] handler
            = new SQLQueriesHandler[{*->*;-<-}]();
        String updateString =
            "INSERT INTO Logs VALUES ";
        updateString += "(NOW(),'" + type + "','"
            + description + "')";
        Console[{*->currentUser;*<-currentUser;
            Sysroot<-*;-<-}] console =
            new Console[{*->currentUser;
            *<-currentUser;Sysroot<-*;-<-}]();

        if(handler.executeUpdate(updateString)){
            console.println("\n\tLogging: " +
                type+ ", " + description + "\n");
            return true;
        }else{
            console.println(
                "\n\tCritical Error: Failed to log.\n");
        }
        return false;
    }
}
```

Listing 4.3: The *Logger* class

```
// class responsible for authentication of HIS users
class AuthenticationManager{

    // find out what roles the user is
    // playing for the patient
    public static principal {Sysroot<-*} role{*<-*}
        (String {Sysroot<-*}userName,
        String {*<-*}patientId){
        ...
        return role;
    }

    // this method returns the username of t
    // he authenticated user
    public static String{Sysroot->_;Sysroot<-*}
        authenticate{*<-*}()
        where caller(Sysroot){

        final label{*<-*} credentialsLabel =
            new label{Sysroot->*;Sysroot<-*};
        ...
        Console[credentialsLabel] console =
            new Console[credentialsLabel]();
        console.print("user-name: ");
        String usr = console.readLine();

        console.print("password: ");
        // hash the password
        String hashPwd = MyUtils.SHA1(console.readLine());
        // here the verification of credentials happens
        ...
        // the userName variable has to be declassified
        return declassify(userName, {Sysroot->_;Sysroot<-*});
    }

}
```

Listing 4.4: Part of the class *AuthenticationManager*

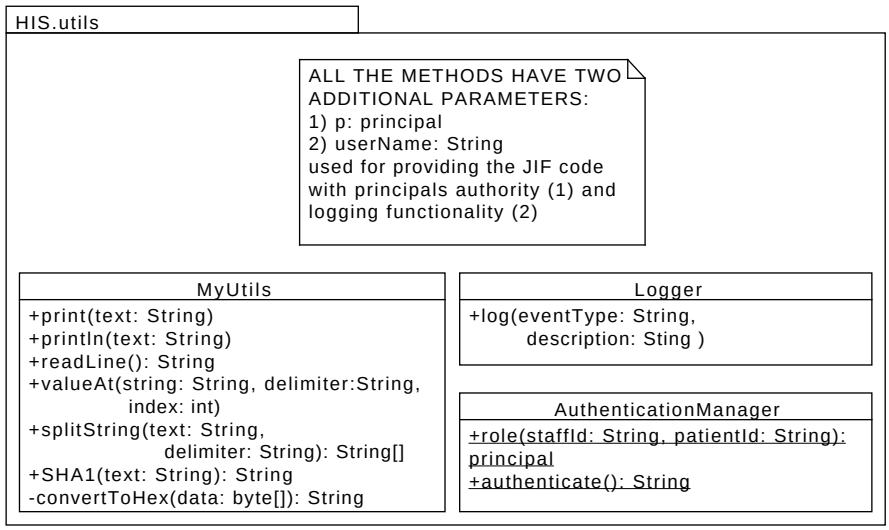

Figure 4.3: The UML class diagram of the *utils* package

4.7 SQL tables implementation

The data handled by the Hospital Information System are permanently stored in a database. In this respect the implementation is very realistic. Many of the information systems interact with databases to read and output processed data. This is also the way the Hospital Information System was implemented.
All the data handled by the Hospital Information System are placed in one database called *hisDB*. The database is composed of a number of tables, as presented in the table 2.7.

Chapter 5

Case study: usage scenarios

In this section we present a number of Hospital Information System usage scenarios, the results of performing them on the the system implemented in JIF and conclusions how is the implementation meeting the design.

The usage scenarios we decided to perform are:

1. adding a patient to the system
2. adding diagnose
3. reading administrative record
4. archiving record

As well as testing functionality , the scenarios have been performed in a way that shows if the implementation is obeying the security policies - we have made attempts to break the policies (e.g. reading medical record by a *Clerk*) and evaluate how the implementation is coping theses attempts.

The mode of testing is functional. We test the implementation of selected, representational use cases. The use cases are specified in section 2.2. Functional

testing belongs to the class of black-box test - what matters for the test result is the visible outcome of the performed test, thus in the test we will focus more on the visible outcome than the internal state of the system.

5.1 Test case: add patient - authorized attempt

This test case verifies the implementation of the functionality of adding a new patient to the system. The test will be executed with the assumption that the user is authorized to perform the operation of adding a new patient to the system.

Table 5.1 presents the test case and the figure 5.1 presents a screen-shot of the test case result.

Test case name:	add patient authorized
Summary:	user of role Clerk adds a new patient to the system
Actors:	user *adm1*
Use case tested:	N/A
Use case variant:	N/A
Pre-conditions:	- user *adm1* is an *Clerk* - *adm1* is authenticated
Scenario:	1. the user *adm1* issues command "addPatient"
Expected result:	a new patient is added to the system
Test result:	Accepted
Comments: -	

Table 5.1: Test-case *add patient authorized*

```
[AdministrativePerson]>     addPatient
                            pat5
                            Jesper Voldby, Mortonsvej 3, 2800 Kgs. Lyngby
                            840423-2315     Anne Voldby,Mortonsvej 3, 2800 Kgs. Lyngby
[AdministrativePerson]>     readAdmRecord
    patient: pat5
    personal data: Jesper Voldby, Mortonsvej 3, 2800 Kgs. Lyngby
    ssId: 840423-2315
    next of kin: Anne Voldby,Mortonsvej 3, 2800 Kgs. Lyngby
```

Figure 5.1: A screen-shot of the test case *add patient authorized*

5.2 Test case: add patient - unauthorized attempt

This test case verifies the implementation of the functionality of adding a new patient to the system. The test will be executed with the assumption that the user is not authorized to perform the operation of adding a new patient to the system.

Table 5.2 presents the test case and the figure 5.2 presents a screen-shot of the test case result.

Test case name:	add patient unauthorized
Summary:	user of role Doctor adds a new patient to the system
Actors:	user *adm1*
Use case tested:	N/A
Use case variant:	N/A
Pre-conditions:	- user *doc1* is a *Doctor* - *doc1* is authenticated
Scenario:	1. the user *doc1* issues command "addPatient"
Expected result:	the user is denied the operation and the event is logged
Test result:	Accepted
Comments: -	

Table 5.2: Test-case *add patient unauthorized*

```
doc1[Doctor]>   addPatient
                pat5
                Jesper Voldby, Mortonsvej 3,2800 Kgs.Lyngby
                840423-2315
                Anne Voldby, Mortonsvej 3,2800 Kgs. Lyngby

        Logging: unauthorized command, user doc1 tried to add a new patient
```

Figure 5.2: A screen-shot of the test case *add patient unauthorized*

5.3 Test case: reading administrative record - authorized attempt

This test case verifies the implementation of the *reading administrative record* use case. The use case will be executed with the assumption that the user is authorized to perform the reading administrative record operation.

Table 5.3 presents the test case and the figure 5.3 presents a screen-shot of the test case result.

Test case name:	read administrative record authorized
Summary:	user of role Clerk reads the administrative record of a patient
Actors:	user *adm1*
Use case tested:	read administrative record
Use case variant:	the user plays role Clerk
Pre-conditions:	- the program is run for patient *pat1* - user *adm1* is an *Clerk* - *adm1* is authenticated - patient *pat1* is registered in the system
Scenario:	1. the user *adm1* issues command "readAdm-Record"
Expected result:	the administrative record of patient *pat1* is displayed
Test result:	Accepted
Comments: -	

Table 5.3: Test-case read *administrative record authorized*

```
adm1[AdministrativePerson]> readAdmRecord
    patient: pat1
    personal data: Anders Jacobsen, Lyngby Hovedgade 34, 2800 Kgs. Lyngby
    ssId: 250375-3215
    next of kin: Nina Jacobsen, Lyngby Hovedgade 34, 2800 Kgs. Lyngby
```

Figure 5.3: A screen-shot of the test case *read administrative record authorized*

5.4 Test case: reading administrative record - unauthorized attempt

This test case verifies the implementation of the *reading administrative record* use case. The use case will be executed with the assumption that the user is not authorized to perform the reading administrative record operation.

Table 5.4 presents the test case and the figure 5.4 presents a screen-shot of the test case result.

Test case name:	read administrative record unauthorized
Summary:	user of role else than Clerk reads the administrative record of a patient
Actors:	user *doc1*
Use case tested:	read administrative record
Use case variant:	regular
Pre-conditions:	- the program is run for patient *pat1* - user *doc1* is a *Doctor* - *doc1* is authenticated - patient *pat1* is registered in the system
Scenario:	1. the user *doc1* issues command "readAdmRecord"
Expected result:	the user is denied access to the administrative record,the event is logged
Test result:	Accepted
Comments: -	

Table 5.4: Test-case read *administrative record unauthorized*

```
doc1[Doctor]> readAdmRecord

    Logging: unauthorized command, user doc1 tried to
             read admnistrative record of patient pat1
```

Figure 5.4: A screen-shot of the test case *read administrative record unauthorized*

5.5 Test case: adding diagnose - authorized attempt

This test case verifies the implementation of the *adding diagnose* use case. The use case will be executed with the assumption that the user is authorized to perform the add diagnose operation.

Table 5.5 presents the test case and the figure 5.5 presents a screen-shot of the test case result.

Test case name:	add diagnose authorized
Summary:	user of role Doctor reads adds a diagnose for a patient
Actors:	user *doc1*
Use case tested:	add diagnose
Use case variant:	regular
Pre-conditions:	- the program is run for patient *pat1* - user *doc1* is an *Doctor* - *doc1* is authenticated - doctor *doc1* is assigned patient *pat1* - patient *pat1* is registered in the system
Scenario:	1. the user *doc1* issues command "addDiagnose" with required parameters
Expected result:	the new diagnose is added to the system
Test result:	Accepted
Comments: -	

Table 5.5: Test-case read *add diagnose authorized*

```
doc1[Doctor]> addDiagnose   flu
doc1[Doctor]> readDiagnoses
    patient: pat1
    diagnose: flu
    author: doc1
    date: 2011-06-05 16:11:01.0
```

Figure 5.5: A screen-shot of the test case *add diagnose authorized*

5.6 Test case: adding diagnose - unauthorized attempt

This test case verifies the implementation of the *adding diagnose* use case. The use case will be executed with the assumption that the user is not authorized to perform the add diagnose operation.

Table 5.6 presents the test case and the figure 5.6 presents a screen-shot of the test case result.

Test case name:	add diagnose unauthorized
Summary:	user of role else than Doctor reads adds a diagnose for a patient
Actors:	user *doc1*
Use case tested:	add diagnose
Use case variant:	the user plays role Nurse
Pre-conditions:	- the program is run for patient *pat1* - user *nurse1* is an *Nurse* - *nurse1* is authenticated - doctor *nurse* is assigned patient *pat1* - patient *pat1* is registered in the system
Scenario:	1. the user *nurse1* issues command "addDiagnose" with required parameters
Expected result:	the user is denied adding the diagnose, the event is logged
Test result:	Accepted
Comments: -	

Table 5.6: Test-case read *add diagnose unauthorized*

```
nurse1[Nurse]> addDiagnose      flu

    Logging: unauthorized command, user nurse1 tried to add
            a diagnose to the patient: pat1
```

Figure 5.6: A screen-shot of the test case *addd diagnose unauthorized*

5.7 Test case: archiving record - authorized attempt

This test case verifies the implementation of the *archive record* use case. The use case will be executed with the assumption that the user is authorized to perform the archive record operation.

Table 5.7 presents the test case and the figure 5.7 presents a screen-shot of the test case result.

Test case name:	archive record authorized
Summary:	user of role *Clerk* archives the record of a patient
Actors:	user *adm1*
Use case tested:	archive record
Use case variant:	regular
Pre-conditions:	- the program is run for patient *pat5* - user *adm1* is a *Clerk* - *doc1* is authenticated - patient *pat5* is registered in the system
Scenario:	1. the user *adm1* issues command "archiveRecord"
Expected result:	the record is archived
Test result:	Accepted
Comments: -	

Table 5.7: Test-case read *archive record authorized*

```
adm1[Clerk]> archiveRecord

        record archived!
```

Figure 5.7: A screen-shot of the test case *archive record authorized*

5.8 Test case: archiving record - unauthorized attempt

This test case verifies the implementation of the *archive record* use case. The use case will be executed with the assumption that the user is not authorized

to perform the archive record operation.

Table 5.8 presents the test case and the figure 5.8 presents a screen-shot of the test case result.

Test case name:	archive record unauthorized
Summary:	user of role *Nurse* archives the record of a patient
Actors:	user *doc1*
Use case tested:	archive record
Use case variant:	regular
Pre-conditions:	- the program is run for patient *pat5* - user *doc1* is a *Doctor* - *doc1* is authenticated - patient *pat5* is registered in the system
Scenario:	1. the user *doc1* issues command "archiveRecord"
Expected result:	the user is denied archiving the record, the event is logged
Test result:	Accepted
Comments: -	

Table 5.8: Test-case read *archive record unauthorized*

```
doc1[Doctor]> archiveRecord

        Logging: unauthorized command, user doc1
                 tried to archive record of patient pat5
```

Figure 5.8: A screen-shot of the test case *archive record unauthorized*

x

Chapter 6

Case study: benchmark realisation in Aspect Oriented Programming

In this chapter we make a benchmarking of the Decentralized Label Model/JIF against the **Adaptable Access Control** [7] - a recenly published (2009) scheme for providing access control in the area of Electronic Medical Record system.

The Adaptable Access Control (AAC) is strongly based on Aspect Oriented Programming, thus first we will briefly introduce the basic concepts of the AOP - a programming paradigm catering cross-cutting concerns. (AOP)

6.1 Aspect Oriented Programming

In this section we discuss briefly what is the Aspect Oriented Programming and give an example of how it looks.

Aspect Oriented Programming (AOP) is a programming paradigm for separation of cross-cutting concerns. A cross-cutting concern in programming is a functionality that is shared by many features.
A well-known example of a cross-cutting concern are for example undo-redo or logging functionalities.
Many software applications are expected to log the actions of the users (e.g. for audit reasons). It means that in a large number of places in the code some logging functions have to be called. Change of the way the logging is done results normally an overhaul of the whole application - in any place the logging is done. Adding, deleting, or modifying a cross-cutting concern affects major parts of code. For this reason there is a need for a technique that could separate these concerns, and the Aspect Oriented Programming is such a technique.

6.1.1 Basics

Aspect Oriented Programming allows to specify cross-cutting functionalities only in one place, and whenever one of these functionalities is needed, it is called. The difference between AOP and abstraction of functionalities in software components (structured programming) is the way of specifying when a cross-cutting functionality is called. In the structured and imperative programming paradigms (broadly used programming paradigms in software development now-a-days) separation of concerns is achieved by means of encapsulation in classes (or components) and procedures, respectively.
Continuing the example of logging functionality, in the structured programming paradigm the logging functionality could be encapsulated in a logger class (see an example in listing 6.1), and whenever and action is to be logged, the action code would invoke methods of the logger class. In the imperative programming paradigm, the logging functionality would be specified in a set of procedures, similarly as in the case of structured programming (see an example in listing 6.2).

```
class ActionLogger{
    ...
    public static boolean addEntry(
        Action action, Object[] parameters){
        // creating a log entry from the input parameters
        ...
        DatabaseConnection dc = new DatabaseConnection();
        dc.execute(actionLogInsertStatement)
        ...
    }
}

// To provide the logging functionality of actions, each action
//should make a call of the addEntry method, like this:
...
public boolean actionInsertElement(
    Object element){
    ActionLogger.addEntry(
        new Action("InsertElement", element));
    ...
}
...
// and the same way for all actions...
```

Listing 6.1: Logging functionality in **Java**

```
bool addLogEntry(char* action, void* parameters){
    //creating a log entry from the input parameters
    ...
    return insertIntoDatabase("Logs" , &logText)
}
// To provide the logging functionality of actions, each action
//should make a call of the addLogEntry procedure, like this:

bool actionInsertElement(void* element){
    void* params[1];
    params[0] = element;
    addLogEntry("InsertElement", params)
    ...
}
// and the same way for all actions...
```

Listing 6.2: Logging functionality in **C**

On the other hand in the Aspect Oriented Programming the cross-cutting functionalities are specified in so-called **aspects**. An aspect is a tuple of a set of places in the code where the cross-cutting functionality should be executed (called **join points**) and the functionality code to be executed (this code is called **advice**).

Now some terminology from the Aspect oriented programming:

- **join point** - a point in a program code, where an advice can be executed. Sample join points are
  - method invocation
  - object constructor invocation
  - variable or field assignment
  - variable or field reading
- **point-cut** - a set of join points. The point-cuts can be specified generically, e.g. by patterns - every possible join point is compared against the pattern, and if it matches, it belongs to the point-cut, otherwise not. **Example:** all code points where a method which name starts with string "get" is invoked. All "getters" invocation would comprise this point-cut.
- **advice** - the code of a cross-cutting functionality. An advice can be executed, before, after or around (instead of) a join point.
- **aspect** - a 2-tuple of point-cut and advice. It specifies which advice and when should be executed.

```
//logging functionality with AspectJ
aspect Logger{
    //Logger object for actions
    private Logger logger = new Logger(LogType.action);

    // all the join points where the logging should be
    // done - whenever a method with name starting
    // with "action" is called
    pointcut actionCall(Object[] actionParams):
        call(* *action*(Object[])) && args(actionParams);

    //advice to be executed after an action
    after: actionCall(Object[] actionParams)
    {
        //actionParams[0] is action name
        logger.addLog("action call: " + actionParams[0]);
    }
}
```

Listing 6.3: Logging functionality in **AspectJ language**

Usually in Aspect Oriented Programming cross-cutting concerns are built on top of easier separable functionalities - that is basically the idea of this programming paradigm. The easily separable functionalities are implemented in other programming paradigms (e.g. structured or imperative programming paradigms), an then the cross-cutting concerns are intertwined into the base code. This process is called **weaving** and is done by special programs called **aspect weavers**.

6.1.2 Aspect Weavers

An aspect weaver takes as input the non-cross-cutting functionalities code and the aspects. For each join point the weaver inserts the advices of the aspects which point-cut includes the join point. The weaving is a pre-compilation operation. The advices code is added to the base code before the latter is compiled. This way the weaving process does not induce any runtime errors - except the ones already existing in advices. This is an important fact for the use of Aspect Oriented Programming for ensuring data security and providing access control mechanism in software systems, in particular in case of an Hospital Information System.

6.2 Adaptable Access Control and the Hospital Information System

In this section we will investigate how the Adaptable Access Control could be used to meet the security requirements of the Hospital Information System.

Data security and access control are indeed cross-cutting concerns. If a software system is expected to protect data confidentiality and integrity, it should do it throughout the whole system. Coming back to the Hospital Information System, it is not good enough to protect the patient's private informations only in one component of the system. Protection of sensitive data has to be ensured in the whole system. However without a formal method like model checking,static or dynamic program analysis, protection of data confidentiality and integrity is not very reliable. Code control and inspections performed by humans do not give any formal premise to conclude that some security properties hold in a software system.
Aspect Oriented Programming could be called "the third way" - it is more formalized, automatized and deterministic than code reviews. If the weaver is correctly implemented and the weaving process goes correctly (can be verified formally) - the resulting software code may be guaranteed secure in some respects. This is an important property for the Hospital Information System. It has to be proven secure.

The Adaptable Access Control is an access control mechanism designed to protect data confidentiality and integrity in Electronic Medical Records. The authors designed and implemented following:

1. a syntax for specifying data security policies for "tree structured data"
2. an XML schema ("Access Control rules in XML") and parser for these security polices
3. a syntax for specifying a mapping between the variables / methods names in a system implementation and their aliases used in the policy file
4. an XML schema ("Application Specification") and parser for those mappings
5. a set of abstract aspects templates for aspect code generation, and the aspect code generator itself ("Access control rule translator")

The figure 6.1 presents the work-flow in the Adaptable Access Control. The security policies and and the application mapping are input files for the aspect code generator. The generator produces "Access control aspects" - code of aspects that will be weaved into the EMR application (here assumed to be a "Struts-based Web EMR application ")

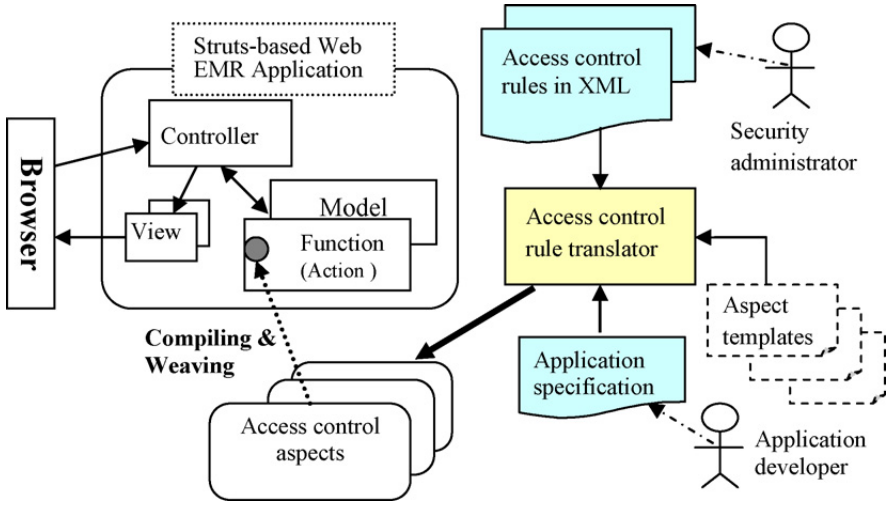

Figure 6.1: Work-flow in the Adaptable Access Control [7]

6.2.1 Adaptable Access Control policies for the Hospital Information System

The analysis performed in the "Gathering requirements" section (1.3) has been concluded with a listing of security requirements for the Hospital Information System. Now we will try to specify a Adaptable Access Control policy file embracing those requirements.

The listing 6.4 presents the policies specified in the AAC policy file format. This policy file includes all access control policies that has been implemented in the Hospital Information System using the JIF language.

The authors of the Adaptable Access Control did not publish the software implementing the "access control rule translator" neither they have shown how to specify the mapping ("application specification") thus we could not really

implement a system and test it. However the authors did make a prototype application which has proven secure [7], thus we assume that it is possible to implement the Hospital Information System and intertwine the access control mechanism in it. Because of the fact that no actual implementation was done using the Adaptable Access Control we will restrain from comparing the JIF and Adaptable Access Control with respect to time needed to develop a Hospital Information System using either technology.

```
<<  PatientRecord, PWD # authentication by username-password pair

    /MedicalRecord::
        Create:
            ("AdministrativePerson" in User.roles);
        Read,Update:
            ("Doctor" in User.roles) and
            (User.name = Data.caringDoctor or
                Context.isEmeregency);
        Archive:
            ("AdministrativePerson" in User.roles);

    /MedicalRecord/currentMedicine:
        Read:
            (
                ("Doctor" in User.roles) and
                (User.name = Data.caringDoctor or
                    Context.isEmeregency)
            ) or
            (
                ("Nurse" in User.roles) and
                (User.name = Data.caringNurse or
                    Context.isEmeregency)
            );
        Update:
            ("Doctor" in User.roles) and
            (User.name = Data.caringDoctor);


    /Diagnoses::
        Add:
            ("Doctor" in User.roles) and
            (User.name = Data.caringDoctor);

    /AdministrativeRecord
        Create, Read, Update:
            ("AdministrativePerson" in User.roles);

>>
```

Listing 6.4: The security policies for Hospital Information System in Adaptable Access Control format

Chapter 7

Comparison & Conclusions

In this chapter we list the constraints of the JIF language and compare the JIF against the Adaptable Access Control according to the ability to meet the security requirements for the Hospital Information System. On this basis we evaluate the JIF and present the result of this evaluation in the section 7.3.3.

As the authors of the Adaptable Access Control claim an actual implementation of the scheme, we decided to focus more on evaluating JIF than Decentralized Label Model, as JIF and Adaptable Access Control belong together on the same abstraction level.

First of all we should note, that the JIF language, which is a framework for implementing systems based on the Decentralized Label Model, does not present the flexibility and interoperability of the latter. In the papers demonstrating the Decentralized Label Model [23, 24, 22], the security policies (labels) are truly distributed and dynamic. It is proposed there to label various distributed data sources and make dynamic checks of the labels. This conceptions is very attractive, and if implemented, may be really powerful and effective. However the implementation of the Decentralized Label Model in the JIF language (version 3.3.1) does not ensure these properties.

7.1 Comparison - JIF disadvantages

In the following sections we will present the most important constraints of the JIF that obstruct implementing a real-life EMR system in this language.

7.1.1 Threaded model not supported

A software system implemented in JIF have to be stand-alone programs. The whole system has to be implemented as single deployment component. Currently JIF does not support the threaded model, thus accepting communication from other software components over a network is not possible. Creating network sockets for multi-client data exchange requires the threaded model [3]. It is only possible to implement simple one-to-one communication channels, which is not sufficient for a systems that is expected to receive queries from many users and external systems simultaneously.

Lack of the threaded model makes it also impossible to implement a Graphical User Interface for JIF programs. The times of command prompt interfaces for end-users have already passed long time ago. The modern user interfaces are graphical. Especially a system like Hospital Information System should provide a graphical user interface of great **usability**. This software attribute is very important for the Hospital Information System, as an unclear and obscure user interface may badly affect the perception of the medical staff, and cause misinformation.

The Adaptable Access Control access control mechanism can be built on top of any Java program, in particular one using the threaded model.

7.1.2 Hard-coded security policies (labels)

The labels containing the security policies have to be hard-coded into the JIF programs and thus they have to be known fully in advance. It is not possible to create and manipulate a label in the runtime. It has to be specified before the program is compiled.
Unfortunately the developers of JIF decided to go more in the direction of static program analysis, which blocked the path of developing a dynamic labels management. The dynamics of labels in the current JIF implementation is basically

restricted to dynamic labels comparison (checking if one label is more restrictive than another one in the runtime). This is way too little for implementation of a software system in which the security policies are a subject of changes. Taking the example of the Hospital Information System, if for example there was a new policy for accessing the Administrative Record, saying that doctors now have to access the Social Security ID (for some legal reason), then the Hospital Information System maintenance people would have to dig into the JIF code and change the labels in there. This is unquestionably not realistic as we shall not expect them to know JIF, or even **java** for that matter.
It would be very neat if the dynamics of the labels was more elaborated in the future releases of JIF. The current state of matters excludes JIF as an implementation language for a real-life Electronic Medical Record system.

The Adaptable Access Control allows to specify the policies in one file, which is a subject of modification. This significantly improves the maintainability of the software. If the policies for the data processed by the system change, a security administrator changes only the policy file and re-compiles the whole application. This is way easier and less time consuming than diving into code of a complex and obscure programming language. Re-compilation and re-deployment of a program supported by Adaptable Access Control is a major issue for the system availability , which is crucial for EMR systems. However change of the security policies in JIF programs requires this same action, thus we can say that JIF is not a winner in the flexibility of policies management comparison.

7.1.3 Primitive and hard-coded principals

Most of the comments on poor label dynamics applies to the principals. A principal is a construct used to abstract a user category or role in JIF. Creation and manipulation of JIF (v 3.3.1) principals in runtime is not possible.
Also the way the hierarchy of the principals (in the sense of act-for relationship) is obscure. There is no easy built-in language constructs (or software) for managing the principals hierarchy in the runtime. However it is a bit better than in the case of managing labels, as it is possible.

In the Adaptable Access Control the users can assume various roles that are connected to some privileges in the system. Furthermore the roles a user is assuming may change in the runtime without a need for a re-compilation of the program. The user roles and other attributes are available in the policy file and in the runtime through *User* object. This feature of Adaptable Access Control makes it superior to JIF in respect to user roles management.

However there is no (explicit) support in Adaptable Access Control for user or roles hierarchy. It is not mentioned in [7] how to specify the hierarchy of roles. Perhaps this feature could also be supported by the *User* object.

7.1.4 Missing I/O libraries

The current version of JIF libraries poorly supports the the basic (like reading from a console or a file) and more elaborated (e.g. SQL databases) input/output operations. In order to get these working, the developer has to make a "hack" (called so by JIF developers), i.e. create JIF signature classes and implement them in Java. A signature class is a class that contains only signatures of fields and methods, but not bodies of these.
Knowing this technique, it does not take much time to implement it, but first one has to learn about. However the more important fact about the "hack" is that it introduces a serious threat for the security of the whole application. The signature classes are excluded from label checking, as they contain only signatures of the constructors, methods and variables . Anything flowing into the classes implemented using the "hack" gets out of control - it is not a subject of the label checking mechanism. Similarly the values coming out of methods in these classes can be assigned any security label (in particular, the least restrictive label). One needs to pay careful attention when using the "hack".

The access control mechanism imposed by Adaptable Access Control is built on top of existing code providing the functionality of the system. This functionality most likely include also input/output operations. Thus the communication between the software components and with other software systems depends solely on the framework of the underlying implementation, which is the Java platform for current implementation of Adaptable Access Control. In this respect the Adaptable Access Control is more flexible than JIF.

7.1.5 Long time of learning JIF

This project lasted for 5 months and that was also the time necessary to master the JIF language to an extend that allowed to implement the presented system prototype. The author has already had a strong background in static analysis of code and in formal verification of programs.
Such a long time of learning the language would probably deter a company from employing JIF in implementation of an EMR system. The reason for such a long learning time is that if one wants to implement even a simple security-

enabled program he has to possess the knowledge on virtually any feature of the language. Also the JIF compiler error messages are not easily understandable and it takes some time to deduce what they are really telling.

Learning how to specify a security policy file takes a few hours (including reading the paper about Adaptable Access Control). It is hard to say how much time it takes to specify the "application specification" - mapping between variable and function names to aliases used in the policy file, as the authors of Adaptable Access Control did not explain it nor they gave information how long it could take. Anyway the time for understanding the concepts and features of Adaptable Access Control is many times shorter that the time for learning Decentralized Label Model and JIF.

7.2 Comparison - JIF advantages

Despite the mentioned serious drawbacks, the JIF language is superior to Adaptable Access Control in some respects.

7.2.1 A formal method

The Decentralized Label Model and JIF are have got a sound scientific background. The lattice model by Denning & Denning [11], relabelling rules [24] and the specification of the JIF language [21] provide proofs of correctness for the entire framework. This fact is important from the legal point of view - if a contract for development of an Electronic Medical Record system states that protection of the patient record has to be proved formally (or at least semi-formally), then JIF fulfils this requirement.
Also in case of a data leak, both the software provider and the customer (hospital managers) can argue that it has not happened due to poor security properties of the system.

The Adaptable Access Control does not accommodate this need. This access control mechanism is operations-based - the security checks are done on the method level. Misspecification or underspecification of the "application specification" (the mapping between the methods names and their aliases in the policy file) or incorrect specification of the policy file may result in data leaks. This framework does not provide any formal proofs of security properties.

7.2.2 End-to-end security

Another exceptional property of the JIF is that it provides end-to-end security. The JIF verifier checks the information flow in the entire application, and it is conservative in the checks - if there is a doubt that there may possibly happen a data leak, it complains about it and prevents compilation of the program. The checker takes into account all possible data flows and by that it guarantees formally that if a program has gone through the verification process, it is safe with respect to the policies specified in the code.
Thus if the policies for the data circulating in the system are specified according to the goal security properties of the system, the program will ensure these properties if it gets through the verification process.

As mentioned before, the Adaptable Access Control is method-based - it only checks if a user is allowed to execute a method. What happens next with the data returned by the method is not relevant for Adaptable Access Control. Potentially it is a major threat for data confidentiality and integrity. If the Adaptable Access Control policy file is not consistent with the rules who can access what information, or if there exist in the program a data flow allowing read-ups or write-downs (violation of data confidentiality) this may lead to serious data leaks.

7.3 Conclusions

7.3.1 Decentralized Label Model as a framework for implementing Hospital Information System

Examination of JIF on the Hospital Information System case study has shown that in the current shape is not a proper framework for implementation of Electronic Medical Record systems.
The main issues of JIF are low flexibility, small number of libraries supporting interoperability and obscurity of the language. The current version (3.3.1) of JIF does not seem appropriate for implementation of large systems used for presentation and simple manipulation of data. It is good for security-critical code and complex processing of data with different confidentiality/integrity levels and in such applications it is very powerful.
The need of hard-coding security policies and user roles hierarchy makes programs written in JIF hard to maintain and extend. And as the policies for patient record protection are a subject of constant management, the system

maintenance team has to be skilled in JIF, and that cannot be expected from system administrators of a hospital computer system.
There were not many attempts to scrutinize JIF, in particular on a case study of an Electronic Medical Record system. This facts obstructs growth of a developers community and development of techniques that may be generic for this type of systems.
On the contrary, the Adaptable Access Control, a framework for ensuring access control in Electronic Medical Record systems satisfies most of the requirements for Electronic Medical Record systems. The authors of the Adaptable Access Control implemented a prototype system for management of patients records following an official Taiwan standard for patient records [7].
If the future releases of JIF tackle the issues mentioned in this chapter, possibly JIF will fit more the into Electronic Medical Record system scenarios.

7.3.2 Related work

There has been only a few middle- to large-scale applications developed in the JIF and published:

- SIF - Servlet Information Flow - "a novel software framework for building high-assurance web applications, using language-based information-flow control to enforce security" [9]
- SWIFT - "a new, principled approach to building web applications that are secure by construction. ... Swift automatically partitions application code while providing assurance that the resulting placement is secure and efficient." [8]
- Fabric - "a new system and language for building secure distributed information systems. It is a decentralized system that allows heterogeneous network nodes to securely share both information and computation resources despite mutual distrust. Its high-level programming language makes distribution and persistence largely transparent to programmers." [18]
- Civitas - "the first electronic voting system that is coercion-resistant, universally and voter verifiable, and suit- able for remote voting." [10]
- JPMail - an e-mail client developed at Penn State University (USA) [4]

Among these only the JPMail project strived to scrutinize the Decentralized Label Model and the JIF language. The others were more oriented on the new

concepts introduced in these projects and did not try to take a critical look on JIF. This is quite self-explanatory as the co-authors of these project were the contributors of the JIF language.
The JPMail project aimed both at development of a real-world application and development of additional tools / mechanism for JIF supporting the former goal. The conclusions drawn from JPMail implementation process were following:

- "it is possible to implement a practical application, an e-mail client, in Jif" [4]
- "one challenge is in managing principals beyond the limited domain of a single Jif program execution" [15]
- "better development tools are essential for making security-typed application development practical" [16]
- "despite the substantial amount of work involved, mail client is neither flashy nor full-featured" [15]

These conclusions are in accordance with the conclusions drawn from implementation of the Hospital Information System, however they are sparse and on a high level of abstraction.

An attempt to implement an EMR system using JIF was done by E. Nodet [25]. This work was useful for the Hospital Information System project, as it signalises some of the JIF drawbacks. However lack of explicit goals and criterion for scrutinizing the JIF language and simplicity of the implemented system did not give sufficient information on how well-suited is the Decentralized Label Model and JIF for implementing Electronic Medical Record systems.

7.3.3 Further research

Scrutinization of the JIF language implementation revealed a number of serious drawbacks, however they could be possibly avoided or eliminated. An interesting direction of employing the Decentralized Label Model would be implementation of another framework (not necessarily a programming language) realising the full potential of the model. Key development goals of this new framework should be: flexibility,usability and maintainability of programs using this framework. Pursue of these goals would require strong improvement of labels dynamics, elaboration of the API for principals hierarchy management and shortening the technology learning process.

Appendix A

HIS

A.1 HIS Implementation Reference

A.1.1 Class *Main*

The class *Main* is the entry point of the whole program. The method *main* of this class is the first method invoked when the program is launched.

The class (and the method *main*) has got the authority of **principal *Sysroot*** who can act for all other principals. This declaration of authority is necessary for running the program with authority of the user who has successfully logged in. In the main method there is a call to the *AuthenticationManager* class to authenticate the user. If the authentication is successful, the *CommandProcessor* with authority of the logged in user is instantiated and a session is started.

Methods

main

the main method receives as a parameter the **patientId** of the patient for whom the session is started. Whenever launching the program, **patientId** must be specified, as the user always has to assume a role (e.g. Doctor, Nurse etc.) which may be different for a different patients.
The method first authenticates the current user, then it yields the role of the authenticated user according to the input **patientId**. After that a session for the authenticated user is started and command prompt is launched.
The program terminates when the method terminates.

parameters :

- args: String[] - command line parameters - here the **patientId** is conveyed

A.1.2 Class *MedicalRecord*

Fields

***doctorLabel*: label**

This variable holds a label a part of medical record is labelled with - the data that should be accessible for the **Doctor** only.
The label of the field is $\{\top \leftarrow \top\}$.

***nurseLabel*: label**

This variable holds a label a part of medical record is labelled with - the data that should be accessible for the **Doctor** and the **Nurse**.
The label of the field is $\{\top \leftarrow \top\}$.

***patientIdLabel*: label**

This variable holds a label of the **patientId**.
The label of the field is $\{\top \leftarrow \top\}$.

***patientId*: String**

This variable holds the id of the patient who is the subject of the record.
The label of the field is $\{*patientIdLabel\}$

***currentMedicine*: String**

The medicine the patient is currently treated with.
The label of the field is $\{*nurseLabel\}$

***allergies*: String**

The allergies the patient is suffering.
The label of the field is $\{*nurseLabel\}$

***specialNote*: String**

Important patient-specific information, e.g. mental diseases, handicapped.
The label of the field is $\{*doctorLabel\}$

Methods

create

This method adds a new medical record entry to the system.

parameters :

- patientId: String - id of the new patient added to the system

load

This method loads from the database the medical record of a patient.

parameters :

- patientId: String - id of the patient whose record is to be loaded

update

This method uploads the medical record to the database.

parameters :
this method takes no parameters

print

This method prints the medical record to the standard output.

parameters :
this method takes no parameters

assignMedicine

This method prints the medical record to the standard output.

parameters :

- medicine: String - name of the medicine to be assigned

appendAllergy

This method adds information about a newly discovered allergy of the patient.

parameters :

- allergy: String - name of the allergic substance

writeSpecialNote

This method allows to insert the special note about the patient to the system.

parameters :

- note: String - the special note about the patient

A.1.3 Class *AuthenticationManager*

The *AuthenticationManager* class is responsible for authenticating the users and yielding their roles in the system.

Fields

***authenticationLabel*: label**

This variable holds the label the credentials are labelled with - these data should be accessible for the **Sysroot** only.
The label of the field is $\{\top \leftarrow \top\}$.

Methods

authenticate

The *authenticate* method is responsible for authenticating the user - checking if the credentials provided by the user are correct.

parameters :
this method takes no parameters

returns :
the return type is *String*. The return value is the name of the user that has provided correct credentials.

role

The *role* method is responsible yielding the role of the user.

parameters :

- staffId: String - staff id of the user
- patientId: String - id of the patient for whom the role is played

returns :
the return type is *Principal*. The return value is the role the **user** is assuming in the system.

A.1.4 Class *Logger*

The *Logger* class is responsible for logging security-related events in the system. This includes e.g. attempts of unauthorized data access or emergency access of patient record.

Methods

log

This method insert a log entry to the logs database.

parameters :

- eventType: String - string describing the type of event (e.g. "unauthorized access attempt")
- description: String - a more verbose description of the event (e.g. who was the user, what data was accessed etc.)

A.1.5 Class *Console*

The *Console* class facilitates standard input and standard output operations: reading from and writing to the command prompt.

Special Remarks

In the **JIF** library there exist a class *Runtime* intended to provide similar functionality. Unfortunately due to implementation failures it does not work properly and could not be used in the system. In order to facilitate standard input and standard output operation in the *Console* class there was a need to create a **JIF** signature class and implement it in Java, as use of the standard *System.in* and *System.out* streams is restricted in the **JIF** library, in fact it is not allowed.

Methods

print

The *print* function prints a string to the standard output.

parameters :

- text: String - the string printed out to the standard output

println

The *println* function prints a string + a newline character to the standard output.

parameters :

- text: String - the string printed out to the standard output

readLine

The *readLine* function reads a *String* from the standard input.

parameters :
this method takes no arguments

returns :
the return type is *String*. The return value is the string read from the standard input.

A.1.6 Class *Assigned Staff*

Fields

***patientId*: String**

Id of the patient to whom the staff is assigned.

***doctorStaffId*: String**

Staff id of the doctor who is assigned to the patient.

***nurseStaffId*: String**

Staff id of the nurse who is assigned to the patient.

Methods

create

This method adds a new staff assignment entry to the system.

parameters :

- patientId: String - id of the patient for whom the staff assignment entry is added

load

This method loads from the database the staff assignment to a patient.

parameters :

- patientId: String - id of the patient to whom the staff is assigned

update

This method uploads the staff assignment to the database.

parameters :
this method takes no parameters

print

This method prints to the standard output the staff assignment.

parameters :
this method takes no parameters

print

This method prints to the standard output the staff assignment.

parameters :
this method takes no parameters

assignNurse

This method assigns a nurse to the patient.

parameters :

- nurseStaffId: String - staff id of the nurse who is being assigned to the patient

assignDoctor

This method assigns a doctor to the patient.

parameters :

- doctorStaffId: String - staff id of the doctor who is being assigned to the patient

A.1.7 Class *CommandsProcessor*

Fields

***currentUser*: Principal**

The field *currentUser* contains information what role current user is assuming (e.g. Doctor, Nurse etc.)

***currentUserId*: String**

The *currentUserId* is the staff id of the current user.

***patientId*: String**

The id of the patient for which the program is run.

Methods

addDiagnose

This method calls the *Diagnoses.addDiagnose* method to add a diagnose to the patient's record.

parameters :

- parameters: String[] - parameters for the *Diagnoses.addDiagnose* method

addPatient

This method adds a new medical record, administrative record, and staff assignment record for a new patient.

parameters :

- parameters: String[] - parameters used for creating the patient record

addTreatment

This method calls the *Treatments.addTreatment* method to add a treatment to the patient's record.

parameters :

- parameters: String[] - parameters for the *Treatments.addTreatment* method

archiveRecord

This method serves for archiving the patient's record.

parameters :

- parameters: String[]

assignDoctor

This method calls the *AssignedStaff.assignDoctor* method to assign a **Doctor** to the patient.

parameters :

- parameters: String[]

assignNurse

This method calls the *AssignedStaff.assignNurse* method to assign a **Nurse** to the patient.

parameters :

- parameters: String[]

createUser

This method allows to add new users (staff members like Doctors etc.) to the system.

parameters :

- parameters: String[]

listPatients

This method allows list all patients enrolled in the system.

parameters :

- parameters: String[]

readAdministrativeRecord

This method prints to the standard output the administrative record of the patient.

parameters :

- parameters: String[]

readAssignedStaff

This method prints to the standard output staff ids of the staff members that are assigned to the patient.

parameters :

- parameters: String[]

readDiagnoses

This method prints to the standard output the diagnoses that were made to the patient.

parameters :

- parameters: String[]

readMedicalRecord

This method prints to the standard output the medical record of the patient.

parameters :

- parameters: String[]

readTreatments

This method prints to the standard output the treatments the patient has undergone.

parameters :

- parameters: String[]

printHelp

This method prints to the standard output the help information how to use the program.

parameters :

- parameters: String[]

printSyntaxError

This method prints to the standard output an error that the user has mistyped a command.

parameters :

- parameters: String[]

startSession

This method starts a command prompt where user can type-in commands.

parameters :

- parameters: String[]

A.1.8 Class *Diagnoses*

Fields

***dataLabel*: label**

This variable holds a label the diagnoses are labelled with - the data that should be accessible for the **Doctor** only.
The label of the field is $\{\top \leftarrow \top\}$.

***patientIdLabel*: label**

This variable holds a label of the **patientId**.
The label of the field is $\{\top \leftarrow \top\}$.

***patientId*: String**

This variable holds the id of the patient who is the subject of the diagnoses.
The label of the field is $\{*patientIdLabel\}$

***diagnoses*: String[]**

The diagnoses the patient was given by the **Doctors**.
The label of the field is $\{*dataLabel\}$

Methods

addDiagnose

This method adds a new diagnose to the record of the patient.

parameters :

- diagnose: String - textual description of the diagnose

printAll

This method prints all the diagnoses to the standard output.

parameters :
this method has got not parameters

A.1.9 Class *Treatments*f

Fields

***dataLabel*: label**

This variable holds a label the treatments descriptions are labelled with - the data should be accessible for the **Doctor** only.
The label of the field is $\{\top \leftarrow \top\}$.

***patientIdLabel*: label**

This variable holds a label of the **patientId**.
The label of the field is $\{\top \leftarrow \top\}$.

***patientId*: String**

This variable holds the id of the patient who is the subject of the treatments.
The label of the field is $\{*patientIdLabel\}$

***treatments*: String[]**

The treatments the patient has undergone.
The label of the field is $\{*dataLabel\}$

Methods

addTreatment

This method adds a new treatment description to the record of the patient.

parameters :

- treatment: String - textual description of the treatment

printAll

This method prints all the treatments descriptions to the standard output.

parameters :
this method has got not parameters

A.1.10 Class *Utils*

This class is embraces all the methods that are general for the whole system - they are called from various places in the code. It also includes requisite methods that are accessible in **Java** libraries, but not in the **JIF** libraries.

Methods

splitString

This method splits a string into sub-string on a selected delimiter. It is the same as the standard *String.split* method which is excluded from the JIF library.

parameters :

- text: String - string to be split.
- delimiter: String - splitting delimiter

returns :
the return type is *String*[]. The return value is array of sub-strings of the input string split on the selected delimiter.

SHA1

This method returns a SHA1 hash value of the input *String*. The SHA1 hashing function is generally recognized a secure hashing function.

parameters :

- text: String - string to be hashed.

returns :
the return type is *String*. The return value is the hash value of the input string.

convertToHex

This method converts a *byte*[] array into a *String*. It is used by the *SHA1* method.

parameters :

- data: byte[] - input byte array.

returns :
the return type is *String*. The return value is the input array converted into a *String*.

A.1.11 Class *PatientsList*

This class is used to retrieve the list of all patients enrolled in the system.

Fields

***dataLabel*: label**

This variable holds a label the patients list is labelled with - the data should be accessible for the **Administrative Person** only.

***patientsList*: String[]**

This variable holds an array with the **patientId**'s of all the patients enrolled in the system.

Methods

load

This method populates the patients list from the database.

parameters :
this method takes no arguments

printAll

This method prints all the *patientsList* variable to the standard output.

parameters :
this method takes no arguments

A.1.12 Class *SQLQueriesHander*

This class is used to as an interface between the JIF code and **SQL** databases.

Special Remarks

In the **JIF** library there is no implementation of interfaces to SQL databases. In order to facilitate querying the databases there was a need to create a **JIF** signature class and implement it in Java.

Fields

***rowsNumber*: int**

This variable holds the number of rows of the latest **SELECT** query result.

***columnsNumber*: int**

This variable holds the number of columns of the latest **SELECT** query result.

***databaseName*: int**

This variable holds the name of the database which should be queried.

Methods

getValue

This method return the value of the latest **SELECT** query result at the specific row and column.

parameters :

- row: int - the row number where the value is placed
- column: int - the column number where the value is placed

executeSelect

This method executes a **SELECT** query on the database.

parameters :

- query: String - the **SELECT** query to be executed

executeUpdate

This method executes a query affecting database (**UPDATE**, **INSERT** or **DELETE**)on the database.

parameters :

- query: String - the query to be executed

Bibliography

[1] `http://www.whitcam.com/research/wp-content/uploads/2008/04/taxesgowhere.jpg`.

[2] `http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html`.

[3] `http://download.oracle.com/javase/tutorial/networking/sockets/clientServer.html`.

[4] `http://siis.cse.psu.edu/jpmail/`.

[5] Ab Bakker. Access to ehr and access control at a moment in the past: a discussion of the need and an exploration of the consequences. *International Journal of Medical Informatics*, 2004.

[6] Bob Brown. Protecting the confidentiality of medical records in an interconnected environment. *Journal of Health Care Compliance*, 2010.

[7] Kung Chen, Yuan-Chun Chang, and Da-Wei Wang. Aspect-oriented design and implementation of adaptable access control for electronic medical records. *Journal of Medical Informatics*, 2010.

[8] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*.

[9] Stephen Chong, K. Vikram, and Andrew C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of USENIX Security Symposium 2007*.

[10] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[11] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, July 1977.

[12] Michelle Dougherty. The legal ehr: A new compliance focus. *Journal of Health Care Compliance*, 2008.

[13] Mehrdad Farzandipour, Farahnaz Sadoughi, Maryam Ahmadi, and Iraj Karimi. Security requirements and solutions in electronic health records: Lessons learned from a comparative study. *Journal of Medical Systems*, 2010.

[14] Beatrice Finance, Saida Medjdoub, and Philippe Pucheral. Privacy of medical records: From law principles to practice. In *Proceedings - IEEE Symposium on Computer-Based Medical Systems*.

[15] Boniface Hicks, Kiyan Ahmadizadeh, , and Patrick McDaniel. From languages to systems: Understanding practical application development in security-typed languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference*.

[16] Boniface Hicks, Dave King, and Patrick McDaniel. Jifclipse: Development tools for security-typed languages. In *Proceedings of PLAS07*.

[17] Dimitrios Lekkas and Dimitris Gritzalis. Long-term verifiability of the electronic healthcare records' authenticity. *International Journal of Medical Informatics*, 2007.

[18] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*.

[19] Robert Lowes. Healthcare it: How safe is your patient data? *Medical Economics*, 2006.

[20] G.F. Knolmayer M. Luethi. Security in health information systems: An exploratory comparison of u.s. and swiss hospitals. In *Proceedings of the Annual Hawaii International Conference on System Sciences*.

[21] Andrew C. Myers. Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*.

[22] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 129–142, New York, NY, USA, 1997. ACM.

[23] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels, 1998.

[24] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9:410–442, October 2000.

[25] Emily Nodet. Data-flow control using jif in a health care system. Master's thesis, Technical University of Denmark, 2008.

[26] Brian Regan, O. Tolga Pusatli, Eugene Lutton, and Rukshan Athauda. Securing an ehr in a health sector digital ecosystem. In *2009 3rd IEEE International Conference on Digital Ecosystems and Technologies, DEST '09.*

[27] Lillian Røstad and Øystein Nytrø. Personalized access control for a personally controlled health record. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[28] Richie Saville. The doctor will see you now. *Computer Fraud and Security*, 2010.