Norwegian University of
Science and Technology

Specification of security properties by JML

Ilir Dulaj

Master in Security and Mobile Computing
Submission date: July 2010
Supervisor: Danilo Gligoroski, ITEM
Co-supervisor: Hubert Baumeister, Technical University of Danmark

Norwegian University of Science and Technology
Department of Telematics

Problem Description

This thesis proposal evaluates the feasibility of the Java Modelling Language (JML) for specifying security properties. It investigates possible JML improvements for specifying security properties like confidentiality (only authorized people are allowed to view data), access control (only authorized people have access to data of their domain) etc.
To this end a case study model application for a medical clinic using Java security mechanisms is defined and a risk analysis is conducted deriving security properties.
We examine how well the current JML is able to specify these properties and suggest possible improvements.

Assignment given: 15. February 2010
Supervisor: Danilo Gligoroski, ITEM

Abstract

Nowadays, verification of programs is gaining increased importance. The software industry appears more and more interested in methods and tools to ensure security in their applications. Java Modeling Language has been successfully used in the past by programmers to express their intentions in the Design by Contract fashion in sequential programming. One of the design goals of JML was to improve the functional software correctness of Java applications. Regarding the verification of security properties, JML was mostly successful in Java Smart Card applets due to the specifics of these applications.

In this thesis work we investigate the feasibility of JML to express *high-level* security properties in Java applications that have more realistic requirements and are implemented in the object oriented technology. We do a threat analysis of a case study regarding a medical clinic and derive the required security properties to secure the application. We develop a prototype application where we specify high-level security properties with JML and use a runtime assertion checking tool to verify the code.

We model the functional behavior of the prototype that establishes the security properties as a finite state automaton. Our prototype is developed based on this automaton. States and state transitions modeled in the automaton are expressed in the prototype with JML annotations and verified during runtime.

We observe that currently available features in JML are not very feasible to capture the security related behavior of Java programs on the level of the entire application.

Acknowledgements

I wish to thank my primary supervisor, Professor Hubert Baumeister at Technical University of Denmark for his valuable guidance.

I would also like thank my co-supervisor, Professor Danilo Gligoroski at the Norwegian University Of Science And Technology in Trondheim, Norway for his help.

Special thanks to all the professors and lecturers whom I've met in both of these universities for time and effort spent in educating me during my two years of study.

Table of Contents

1. Introduction 9
   1.1. Motivation 10
   1.2. Objective 11
   1.3. Outline 12
2. Related work 13
   2.1. Specification of applications and applets 13
   2.2. Protocol specification 15
   2.3. Other related work 15
3. Background concepts 17
   3.1. An Overview of JML 17
   3.2. Java RMI 35
   3.3. Authentication and Access Control in Java 38
4. Approaches of using JML to specify security properties 43
   4.1. Correctness of implementation 44
   4.2. State machine representation 51
   4.3. Language Based perspective 55
   4.4. Our approach 59
5. Case Study analysis 63
   5.1. A general overview 63
   5.2. Use cases 66
   5.3. Threat analysis 67
   5.4. Attacker model 69
   5.5. Security goals 70
   5.6. Scope of implementation 73
   5.7. The application 74
6. Implementation of properties 77
   6.1. Architecture of the application 78

6.2. Sequence of operations ... 82
6.3. Modeling the behavior of the application ... 85
6.4. Specification of the program ... 89
6.5. Discussion ... 98
7. Testing and other implementation details ... 101
7.1. Data structure ... 102
7.2. Design of the application ... 103
7.3. Acceptance tests ... 106
7.4. Analysis of the test results ... 111
8. Conclusion ... 113
8.1. Discussion ... 114
8.2. Future Work ... 114

A. **Running the prototype ... 117**
The JAR archives ... 117
Using the assertion checker to verifying the JML annotations ... 118

B. **Source code listings ... 119**
Client.java ... 119
LoginImpl.java ... 123
ServerProxy.java ... 126
ServerImpl.java ... 129
ServerInterface.java ... 132
LoginModule.java ... 133
LoginInterface.java ... 137
MethodcallValidate.java ... 138
ServerPermission.java ... 139
MyPrincipal.java ... 140
MedRecord.java ... 141
Patient.java ... 142
Test.java ... 143
Server.java ... 144

C. **C. Policy and configuration files ... 145**

Authorizations.policy 145
LoginModule.config 146
codePermissions.policy 147

Chapter I

1. Introduction

Today when private companies and governmental agencies gather information on us more than ever, computer security becomes increasingly important for the society at large. Examples of data gathered at large information systems range from credit card numbers, biometric data such as digital fingerprints and the history of patients' illnesses.

At the same time, the need to share this information between relevant actors is nowadays an everyday practice. Examples include, sharing data between the judiciary system and the courts, the insurance companies and the hospitals etc.

Despite numerous benefits from this collaboration, this puts us at greater risk from suffering severe consequences in case of information theft or misuse. To prevent this, special care needs to be dedicated into securing the applications that handle this information.

For this purpose, during the design of applications, software industry employs security experts that formulate security requirements for applications in terms of security properties that those applications must have. Developers then implement these properties.

In order to ensure high degree of security in the application, the software industry uses common techniques such as careful design and testing.

Another way to ensure high quality of the source code is by using some tool that ensures certain properties of the code. These tools statically examine the structure of the code and its interdependencies (1).

Many tools are available, where each has its own specifics and techniques that it uses. One of the tools (FindBugs (2)) enforces certain coding guidelines and uses techniques to detect *errorness coding patterns*. Another tool(ESC/Java (3)) *statically* verifies the code against the provided annotations which are inserted to capture the design decisions that the programmer had in mind during development (3).

Another popular approach especially used for mission-critical applications is *model checking*. A tool like Bandera (4), exhaustively checks a finite state-model of the application for violations of the system requirements (or properties) formally specified in some temporal logic (e.g. LTL[1] (5)). The last technique involves *full verification of the system* by requiring

[1] Linear temporal logic.

both additional annotations with detailed specifications and construction of a proof that the code matches the specifications. The latter task is the most time consuming. Example tools doing full verification are Jack (6), Loop (7) etc.

The number of tools that uses the Java Modeling Language for static analysis and full program verification has been growing in the last decade. These tools rely on JML to specify the intended code behavior in the Design by Contract (DBC) fashion. DBC is based on "contracts" — a formal agreement between a client and its suppliers. To call a method, a client class must satisfy the conditions imposed by a supplier class. If these conditions are satisfied, the supplier class must guarantee certain properties, which constitute the supplier class obligations. On the other hand, if conditions are not satisfied, that is, there is a contract violation, then a runtime error occurs.

It is generally acknowledged that there is a considerable *gap* between properties expressed in this way, by method pre- and post- conditions and class constraints, and the properties and terms that system designers have in mind when they talk about properties in applications.

Even though the JML constructs contribute to ensuring *correctness* that ultimately results in a more reliable software in the long run, they are far from being sufficient in capturing *high-level* properties of applications.

1.1. Motivation

When it comes to specification of security related properties with JML and their verification, the research community has been focusing mainly in Java Smart Card[2] applets (8), (9), (10). Because of the specifics of the hosting environment that has very limited memory and processing capacity, applets are usually *small* programs and have *less* complexity. They *cannot* be compared to a typical Java desktop or web application. The program behavior of these applications is characterized by a compact life cycle. Usually each stage of the life-cycle has precisely determined actions; allowed to be carried during the specific phase.

The resource constraints in their hosting platform dictate specific characteristics and security requirements for these applications (11). The programming language used in Java Cards is a subset of Java with a number of Java constructs absent. It is ideal for application of formal methods, since the language is small enough to be actually feasible for study from a formal perspective, yet it is used in real world applications.

As a result, the specification and verification of security properties with JML in these programs has been quite successful in the research community (12).

In the recent years much research effort[3] has been put in JML by research groups from around the world. Many *new* constructs were added to JML and different case studies done on how to use JML for different verification purposes including security features.

[2] A typical example of Java Cards are SIMs in mobile phones.
[3] Papers regarding JML are available at http://www.eecs.ucf.edu/~leavens/JML/papers.shtml

Considering the success of JML in the Java Card world and the addition of many new semantics to JML in the last years, we were motivated to see how far has this specification language progressed and how useful it is in capturing high level security properties.

In this thesis work our main concern is how JML can be used for security aspects in applications. We have been motivated mainly by the following questions:

- How efficiently can JML be used for specification of larger Java programs that resemble those from real-life scenarios?
- Is it possible to express security properties with JML in these aplications?
- Can we capture *high-level* security properties in the application level by using JML?

1.2. **Objective**

Since JML includes constructs to capture additional aspects of the code behavior, we were interested in exploring how well these constructs are able to specify *security related* behavior of Java applications. Furthermore, we were interested in exploring if it is possible to "push the boundary" of property specification with JML, by aiming at specifying *high-level* security properties such as authentication, authorization etc.

The objective of the work done during this thesis work is twofold. It consists of:

- finding methods how JML can be used to specify *high-level* security properties in Java programs, and
- applying the chosen method(s) to specify security properties on a prototype application developed for a case study regarding a medical clinic.

After researching the JML capabilities, we apply our method for security property specification on a prototype application developed for a case study of our own. The problem definition of the case study that we will work on will be lightly *simplified* in comparison to possible requirements from real life medical clinics.

During our work on the case study we do a threat analysis of the given scenario where we identify security properties required for this application. From the overall group of security properties that we identify during the analysis, we choose a subset of them to implement in the application prototype.

In this prototype, we specify the properties that we have implemented with Java Modeling Language and use a tool verify that there are no violations between the code and the inserted JML statements.

1.3. **Outline**

The reminder of this thesis is organized as follows. Chapter 2 presents related work showing how JML was used in the past to reason about security. In Chapter 3 we provide the background of the concepts that will be used during this thesis work. We give an overview of the main features and limitations of JML. We provide an insight on the RMI technology used during development and the Java security mechanisms that we use during development. In Chapter 4, we present the possible approaches for using JML to specify security properties that were used in the past. We present the approach that we intend to use, listing its similarities and differences with others approaches.

In Chapter 5 we present the scenario of our case study, the threat analysis and the derived security properties from this process. In Chapter 6 we explain how we implement the properties in the prototype and apply our specification method. We present problems and observations during this process. In Chapter 7, we present additional details regarding the implementation and the tests done on the system. Finally, in Chapter 8, we present concluding remarks and point out future works.

Chapter II

2. Related work

In this chapter we present previous work where JML was used to specify properties related to security. The two areas where JML has been particularly successful in specifying security properties are Java Card *applets* and *protocol verification* and less frequently, simple applications implemented in Java.

The software from both of first two categories is usually easy to express in a limited set of precisely defined states which makes them feasible for specification and verification.

Because of the relatively limited size and complexity, JML has successfully been used to specify the behavior of these programs in the past. The provided specifications (which come in the form of annotations) can then be used to statically verify the programs with one of the many tools available that support JML (13).

2.1. Specification of applications and applets

Schubert and Chrzaszcz (14) annotate a simple desktop application for remote password storage. They do a security analysis from the source-code point of view therefore don't take under consideration threats of data storage and threats related to social security attacks. Source-code point of view in this case uses JML to specify a certain behavior of code for which they claim that it meets a set of security properties. This code is then statically verified with the help of static analysis tool of ESC/Java2.

A set of security properties is chosen and features are implemented in the code to counter those threats. These features are then specified using JML[4].

They choose *confidentiality*, *integrity* and *availability* as security properties and list a set of threats against each of these properties. They implement solutions for a number of these threats in the program and specify the code formally using certain JML constructs to be able

[4] Since their goal was to verify the application with ESC/Java2 static analysis tool, besides their own classes, they also had to annotate the behavior of the used Java libraries.

to statically verify that the solutions implemented for achieving those security properties actually work the way they were specified.

A feature implemented in their programe is the analysis and verification of the control flow of the programe. They analyze and express the control flow of the application with the help of *ghost fields*. When a state of interest occurs in the program the `set` keyword is used to modify the ghost field to mark the occurrence of that state in the program.

States expressed in ghost fields are then used in pre- and post- conditions of methods to ensure the general program flow. Object invariants are used to monitor the consistency of objects during the entire execution of the program.

Pre- and post- conditions ensure *context* of the sequence of method calls. They ensure that certain conditions are met before a method is called. For example, when a method call occurs, pre- and post- conditions ensure that certain variables have been initialized. They can ensure that certain objects are created inside the execution of that method and not before it's execution etc.

In their work, they propose a solution to trace the information flow of the confidential data by inserting a ghost field to express the type of data. However they're proposal has one deficiency - it is incapable of tracing the flow of primitive data types.

Despite that they were able to statically verify their application with ESC/Java2 tool, they state that this kind of verification "does not guarantee full security, but provides a certain standardized level of assurance that the source code is well written with regard to the assumed threat analysis".

Jacobs, Oostdijk & Warnier (15) do a case study in formal verification and development of secure Java smart card applications. They specify the control flow of an elementary Java Card applet for a rechargeable phone card. By verifying this flow they are able to prove the correctness of the applet in general. The card life cycle is modeled in finite state machines where program states are marked by ghost fields.

They specify the behavior of cryptographic functions which they use for the challenge-response mechanism in place to ensure secure transactions.

They define high level functional and security requirements of the applet and capture them in the global class invariants and constraints. The functional behavior is captured in the pre- and post- conditions of methods.

In (16) a method for automatically translating high level security properties for applets into JML annotations is shown. The annotations generated by this method are however mainly related to functional correctness of the applet. The first group of properties annotates the *applet life cycle*, describing the different phases that an applet can be in and actions that can be performed in a certain phase. The second group of properties deals with the transaction mechanism ensuring its atomicity. The third group expresses the properties that restrict the exceptions allowed to be thrown and restricts access control of different applications interacting with each other.

Ostdijk and Warnier (17), present how to use JML to annotate applets that use Java Card RMI[5]. They use model variables to specify interfaces that remote objects implement and formally verify the applet with the help of the LOOP (18) tool.

---

[5] Java Card Remote Method Invocation.

2.2. **Protocol specification**

Poll and Schubert (19) present a case study in the specification (and verification) of an open source Java implementation of the SSH protocol. They propose a methodology to formalize this security protocol using finite state machines. JML is used to express the properties they wish to verify for in the source code.

The protocol is represented as a finite state machine which describes how the state of the protocol changes in response to the different messages it can receive. This is of course only a partial specification, as it specifies the order of messages but not their precise format. Nevertheless it is interesting enough because during specification of the implementing code they were able to find security flaws in the implementation. This occurs because specification contributes to a better understanding of the code.

It appears that the specification of this kind is easy to read for non-experts as well. It is beneficial to include in the official protocol specification because it clarifies the specification and makes it easier for developers to implement.

The formal specification of an abstract description for a security protocol for Java smart cards is given in (20). They refine an abstract description of the bilateral key exchange (BKE) protocol into finite state machines. This protocol is described in (21) and it allows two agents to agree on a session key.

JML is used to specify the implementation code and static verification tools are used to verify if the code conforms to the behavior defined in the state machines.

In both of these cases of protocol specification, state-transitions are expressed as a combination of JML invariants, constraints, and method specifications. Static analysis tools are then used to verify if these specifications hold during program execution. If they hold then they assume that the implementation really behaves in the manner it was specified.

2.3. **Other related work**

(22) introduces a security automata as a convenient way to describe security policies. It proposes a translation mechanism for security properties into program annotations. The security properties are expressed as a finite state automaton and annotations are written in JML. The automaton contains a *trap* state to model the occurrence of an error.

The authors define their own semantics of the annotated programs and a few additions to JML to accommodate their semantic constructs.

They use the JML `assert` statement to evaluate a condition on the program state.

The semantics of their language is restricted to a very limited subset of (sequential) Java, excluding typical object-oriented features. The method declarations are limited to containing only one parameter.

(23) uses JML to specify non-interference properties such as *confidentiality* and *integrity* from a language-based perspective. They use a simple security lattice Σ, defined as $\{\mathsf{High}, \mathsf{Low}\}$ with $\mathsf{Low} \subseteq \mathsf{High}$. A secure information flow policy maps variables to security levels and the sets of variables corresponding to those levels.

They're work is based on previous work done by Josh and Leno (24). Confidentiality is guaranteed by restricting the dependency of the fields belonging to the Low confidentiality level from the ones belonging to the High confidentiality level. They use the `\old` keyword from JML to ensure this separation.

Cheo and Perumandla (25) propose an extension to JML to specify the order of method calls. This approach is especially of use when specifying protocols. They propose to separate the protocol assertions from the functional assertions. The specification of the functional properties continues is written in the pre- and post- conditions of methods, but the protocol properties are written directly as separate annotation form proposed by them. For this, they introduce a new sequence clause called `call_sequence`.

The `call_sequence` specification constrains the order in which methods of a class should be called by clients. It specifies the allowed sequences of method calls.

Chapter III

3. Background concepts

This chapter is meant to give a foundation of the concepts used later on during the thesis work. Before we look into how JML has been used in the past to specify security properties, and attempt to do the same in the program of our case study, we explain some of the tools, concepts and programming technologies that we will use during this process.

In this chapter we present a more detailed picture of JML. We begin by presenting some of the most notable features of this specification language, to then continue listing some of it's more advanced features. We list some of the JML limitations that we have identified, when JML is used to specify security properties in Java programs.

We compare JML with another language of a similar purpose, namely with Object Constraint Language (OCL). We point out what are JML's advantages in contrast to OCL, when the design features of the Java programs are specified.

Since the foundation of our case study application is built using the Java Remote Method Invocation technology (RMI), in this chapter, we provide a short description of this programming technology. We show the steps of how to program applications with distributed objects using RMI.

Besides the RMI technology, in our case study we also use Java Authentication and Authorization (JAAS) mechanism. This system authenticates the users based on a built-in authentication mechanism and provides access control depending on their authentication status. In the end of this chapter we explain the architectural components of this system and show how they work together to provide user authentication and authorization.

3.1. An Overview of JML

JML (26), which stands for "Java Modeling Language", is a behavioral interface specification language (BISL) tailored to Java. JML combines the practicalities of design by contract (DBC) principle to specify the behavior of Java classes and interfaces.

During this specification, JML concentrates on two aspects of Java modules:

- syntactic interface – consists of names and static information (e.g., method names, arguments or return type) found in Java declarations;
- functional behavior – describing how the module should behave when used.

JML specifications are written in special *annotation comments*, which start with the @ sign.

Specifications are written in two forms: `//@ <JML specification>` or `/*@ <JML specification> @*/` depending on if we are writing a one line (the first form) or multiple line (second form) specification.

The specification comments are ignored by the Java compiler and interpreted by the JML compiler named JMLC (27). There are numerous languages for this purpose but JML is particularly popular firstly, because its syntax is heavily based on Java (thus easy to use and learn) and, it has a large tool support.

These tools support DBC, runtime assertion checking, discovery of invariants, static checking, specification browsing and even formal verification using theorem provers (28)[6].

As in Eiffel, JML uses Java's expression syntax to write the predicates used in assertions, such as pre- and post- conditions. As a consequence of heavy reliance on Java syntax, JML lacks some expressiveness that makes more specialized assertions difficult to express. To counter this, JML uses various constructs, such as quantifiers.

3.1.1. Assertions and expressions

Assertions and expressions in JML specifications are written using Java's expression syntax. However, they must be pure. This means that side effects cannot appear in JML assertions or expressions.

To prevent side effects, the following Java operators are not allowed within JML specifications:

- assignment — assignment operators like =, += and -= are not allowed
- increment and decrement operators — all forms of increment and decrement operators like ++ and -- are not allowed

In addition to this, only pure methods7 can be used in JML expressions. The purity in JML is expressed by using the `assignable` keyword. Only the fields listed in an `assignable` clause can be modified by a method. Two JML keywords can be used with the `assignable` clause: `\nothing` and `\everything`. We can indicate that a method does not have any side effects by: writing `assignable \nothing`.

[6] JML is a large and very active research project. More documentation, available tools that support JML and background papers are available at: http://www.jmlspecs.org

[7] A method is considered pure is it does not modify the state e.g. by assigning values to any fields or objects.

Below in the method example getName() we show two possibilities of declaring a method as pure – not being able to modify any fields outside their body. However, we should note that they can modify fields declared inside of their body.

```
public /*@ pure @*/ int getName() { return name; }
```

The same can be expressed also by using a one line annotation as listed in Figure 3.1:

```
/@ assignable \nothing;
public int getName( ) {
   return Name;
}
```

Figure 3.1. JML specification with an assignable clause.

This method declaration denotes a method with no side-effects; it is not allowed to modify the program state. The use of the JML modifier pure is equivalent to the assignable \nothing clause.

Besides assignable, JML provides a rich set of constructs, some of which make extensions to the Java's expression syntax to provide more expressive power in JML specification; they can be used in JML assertions and expressions. For example, \old(E) represents the pre-state value of expression E. An expression with a pre-state value refers to the value before method execution. The \result construct specifies the return value of a method. Note that in JML such constructs start with a backslash (\) in order to avoid interfering with identifiers present in a user program.

JML also provides the use of logical connectives such as *conjunction* (&&), *disjunction*(||), *negation* (!), *forward* (==>) and *reverse implications* (<==), *equivalence* (<==>), and *inequivalence* (<=!=>). Regarding quantifiers, JML supports several types such as *universal quantifier* (\forall), *existential quantifier* (\exists), and *generalized quantifiers* (\sum, \product, \min, and \max). The quantifiers \sum, \product, \min, and \max are generalized quantifiers that return respectively the sum, product, minimum, maximum of the values present in JML expressions.

Note also that sometimes during development it is more convenient to insert an *informal description* when specifying a piece of code. Such descriptions are ignored by JML compiler but are there for clarity. They are inserted between (* and *) signs as below:

```
(* some text describing a JML predicate *)
```

As we can see, this description is informal and is entirely left to the reader to interpret it.

In-line assertions

JML provides the use of a specific kind of assertion known as in-line assertions which are specified in the method body. They are bundled together with Java code as shown below in Figure 3.2.

```
public class AssertionExample {
  public void m() {
      //@ assert y1 != 0 && y2 != 0
   y1++;
   y2++;
  }
}
```

Figure 3.2. In-line assertions.

In this example when the execution of method `m()` reaches line:
`//@ assert y1 != 0 && y2 != 0` the assertion must be satisfied or otherwise, an assertion violation is reported.

JML provides several kinds of in-line assertions, such as `assume` statements, `henceby` statements, `unreachable` statements, `set` statements, and `loopinvariant` and `variant` statements.

Default value of declarations

Null pointer exceptions are among the most common faults raised by components written in object-oriented languages such as Java. Because of this, in JML a declaration is by default non-null. So, unless it is explicitly declared by keyword `nullable`, it is required to have a non-null value assigned to it.

Moreover, a class or interface that implicitly contains `nullable` declarations is specified by using the `nullable_by_default` class annotation.

3.1.2. Method specification

In JML, method specifications contain pre-, post- condition predicates. Methods can be specified by *lightweight* and *heavyweight* specifications, normal and exceptional postconditions, and frame conditions. All these features are briefly described.

States of method specifications

JML constructs used for method specifications are divided into three larger groups of states:

- pre-state specifications — these specifications must be evaluated in the pre-state, immediately before the execution of the method body. Example of these specifications are: `requires` clauses, `old` variables and `old` expressions. It is important to note that expressions such as `old` appear in post-state specifications, but they must be evaluated in the pre-state (before method execution);
- post-state specifications — these specifications must be evaluated in post-state, immediately after the execution of the method body. Clauses such a `ensures`, and `signals` are example of post-state specifications;
- internal-state specifications — these specifications must be evaluated in internal states, during the evaluation of the method body. In-line assertions are examples of internal specification state.

Specification clauses

JML provides a number of specification clauses that can be used to specify the behavior of methods (see Table 3.1) the `requires` clauses specifies *preconditions*; the `assignable` clauses specifies *frame conditions*; the `ensures` clauses specifies *normal postconditions*, whereas `signals` clauses specifies *exceptional postconditions*.

`requires`	preconditions
`ensures`	normal postconditions
`signals`	exceptional postconditions
`assignable`	frame conditions

Table 3.1. JML clauses for method specifications.

Preconditions are predicates that must hold before method execution. Figure 3.3 shows an example of a precondition of the method `sum`. According to this specification, the arguments `a` and `b` of the method `sum` must be non-negatives; otherwise an assertion violation is reported.

```
//@ requires a >= 0 && b >= 0;
public int sum(int a , int b) {
   return a + b ;
}
```

Figure 3.3. Example of JML precondition specification.

Normal postconditions are predicates that must hold after method execution without throwing any exception. Figure 3.4 shows an example of a normal postcondition of the method `sum`. According to this specification, the result of the method (indicated by the `\result`

construct) must be equal to the sum of the arguments a and b; otherwise a assertion violation error is raised.

```
//@ ensures \result == a + b;
public int sum( int a , int b) {
  return a + b ;
}
```

Figure 3.4. Example of JML normal postcondition specification.

```
//@ signals (Exception ) a >= 0 && b > = 0;
public int sum( int a , int b) {
// Throws a java.lang.Exception
   return a + b ;
}
```

Figure 3.5. Example of JML exceptional_postcondition specification.

Exceptional postconditions are predicates that must hold when the method terminates by throwing an exception. Each exceptional postcondition can consist of several signals clauses.

In this way, each signals clause must hold when the specified method terminates abnormally by throwing an exception of a type specified in the signals clause. Figure 3.5 shows an example of an exceptional postcondition for the method `sum`. According to this specification, when the method sum throws an exception, the arguments `a` and `b` must be non-negative; otherwise the original exception is intercepted signaling an assertion violation.

Heavyweight and Lightweight specifications

In JML, one can use two different styles on method specifications: *heavyweight* specifications, and *lightweight* specifications (29). A heavyweight specification can start with the keyword `behavior`, which indicates a "complete" specification that includes both normal and abnormal situations. Figure 3.6 shows an example of a behavior specification, including normal (lines 2 to 4), and abnormal situations (line 5).

The `normal_behavior` keyword is used only to define specifications that include only normal situations, that is, no method can terminate abnormally by throwing an exception — the `signals` clause cannot appear in these specifications cases. Figure 3.6 (lines 1 to 3) shows an example of normal behavior specification. Unlike the `normal_behavior`, the

`exceptional_behavior` is employed to specify abnormal situations using the signals clause. Such specifications cannot terminate normally. We emulate this behavior by introducing an implicit `ensures false`.

Figure 3.5 shows an example of an `exceptional_behavior` specification. When an exception is thrown, both input parameters (`a` and `b`) must be greater than zero (lines 1 to 3).

```
/*@ behaviour
@ requires ( a >= 0 ) && ( b >= 0 );
@ assignable \nothing ;
@ ensures \result == a + b;
@ signals (Exception ) a > 200 && b > 100;
@*/
public int sum ( int a , int b ){
   return a + b ;
}
```

Figure 3.6. Example of behavior specification.

```
/*@ normalbehaviour
@ ensures \result == a + b;
@*/
public int soma ( int a , int b ){
  return a + b ;
}
```

Figure 3.7. Example of normal behavior specification.

A heavyweight specification has a well-defined interpretation for each clause in the specification. When one omits a particular clause, JML assumes that it is interpreted as `true`. On the other hand, a lightweight s pecification does not start with a behavior keyword. Thus, only the clauses of interest are specified. In other words, it defines an "incomplete" specification. For example, a method can specify its normal behavior (by defining only precondition or normal postcondition clauses), or its exceptional behavior (by defining only exceptional postcondition clauses). In lightweight specifications, when a particular clause is omitted, the `\not_specified` interpretation is assumed.

The JML semantics states that the keyword `\not_specified` is used to denote that a particular omitted clause has no condition. The JML compiler interprets omitted clauses in lightweight specifications similar to ones in heavyweight specifications which means `true`.

Old expressions

In JML, the keyword `old` is used to refer to pre-state expressions and variables in post-state expressions (e.g., normal postconditions). An old expression, (`\old(E)`), and old variable, (`\old(v)`), denotes the value of the expression `E` and value of variable `v` respectively, in pre-state (i.e. before the method was called).

3.1.3. **Type Specifications**

Invariants are predicates that must hold in all visible (reachable) states[8]. An invariant is annotated in a type declaration by using the keyword `invariant`, also known as *type invariant*. Figure 3.8 shows an example of invariant. In this example, the invariant

(`weight >= 0.0`) must be preserved in all visible states. In this way, if a call to the methods `setWeight` or `getWeight` violates the invariant, an error is raised.

```
public clas s Person {
  public double weight;
  //@ invariant weight >= 0.0;

  public void setWeight (double weight) {
    this.weight = weight;
  }
  public double getWeight ( ) {
    return this.weight;
  }
}
```

Figure 3.8. JML invariant specification.

JML uses the keyword `helper` to provide more flexibility for invariants. The helper modifier (4, Section 7.1.1.14) can only be used on a private method or private constructor. By using this modifier, such methods and constructors are also called *helper methods* and *helper constructors*. In this way, helper methods and constructors are "free" from the obligation of preserving type assertions (e.g., invariants). Suppose that the help method `m` is part of the class Person (see Figure 3.8).

```
private /*@ helper @*/ void m (){ this.weight = -10; }
```

Since it is declared of type helper, it does not need to satisfying the invariant

(`weight >= 0.0`). The example presented in Figure 3.8 refers only to instance invariants. In JML, the invariants are distinguished into static and instance invariants. The former refers only to static fields and methods. On the other hand, the latter can refer to both static and instance fields and methods.

The following states of the program execution must be preserved by an object in order to keep an instance invariant integrity:

- after execution of all non-helper constructors;
- before execution of a non-helper finalizer method;
- before and after execution of all non-helper non-static non-finalizer methods.

[8] Visible states are all the possible states in the program execution.

On the other hand, instance invariants need not be preserved in the following states:

- after execution of constructors declared with the helper modifier;
- before and after execution of methods declared with the helper modifier

```
public clas s TemporaryInvariantBreak {
  public int f;
  //@ invariant f >= 0;
  public void m() {
    this.f = -10; // temporary invariant break!
    System.out.println("internal -state");
    System.out.println("f:" + f) ;
    this.f = 0; //re-establishment of the invariant
  }
}
```

Figure 3.9. Example of temporary invariant violation in a JML specification.

An invariant may be explicitly declared to be a static or an instance by using the modifiers `static` or `instance` during the declaration of the invariant. If an invariant is declared in a class without the modifier, it is an instance invariant by default, whereas if an invariant is declared in an interface without the modifier, it is an static invariant by default.

Regarding invariant violations in JML, there is only one way to temporarily break an invariant that consequently throws no assertion violation error. This temporarily invariant break is established in an internal-state during evaluation of a method's body. However, before the method returns to the caller, the invariant must be reestablished, that is, a method during the execution of its body can break an invariant many times, but before the end of its execution all invariants must be reestablished, otherwise an error is raised indicating the invariant assertion violation. The example in Figure 3.9 demonstrates the temporarily violation of an invariant assertion even in presence of callbacks (recursive assertion checking). In this example the class `TemporaryInvariantBreak` has an invariant condition (line 3) that refers to the field `f` (line 2). For the invariant break purpose, the class `TemporaryInvariantBreak` has a method `m` (lines 5 to 10) that temporarily breaks the invariant declared in line 3. The invariant break occurs in line 6 of the method m. Once broken, one can use the values of the internal-state (line 8 is printed the value of field `f`). However, before the method terminates its execution, the invariant is established again (line 9) in order to prevent invariant assertion violation. Regarding invariants and method termination, methods and constructors must preserve invariants in the case of normal and abnormal terminations. For example, even if a method terminates abnormally by throwing an exception, the invariant check must be performed.

Methods and constructors must preserve invariants in the case of normal and abnormal terminations. For example, even if a method terminates abnormally by throwing an exception, the invariant check must be performed.

An invariant can be declared with any Java's access modifiers (`private`, `protected`, and `public`). As in class members, if an invariant is declared in a class with no one of the Java

access modifiers, this means that it has a package visibility (default access). An invariant declared in an interface with no modifiers has public visibility.

History constraints

History constraints, known as *constraints* for short, are used to specify the way that objects can change their values from one state to another (i.e., pre-state to a post-state) during the program execution.

In JML, a constraint is written in a constraint clause with the keyword `constraint`. Moreover, constraint clauses are usually written using `old` expressions to relate the previous state (pre-state) with the resultant state (post-state), because they are two-state predicate. On the other hand, invariants (discussed in a previous section) cannot use old expressions, because they are *one-state predicates*. The constraints are applied to methods, and they must hold in the post-state of every (non-helper) method execution.

```
public class ConstraintExample {
  public int index ;
  //@ constraint index == \old (index + 1);
  public void incIndex () {
     index++;
  }

  // . . . other methods
}
```

Figure 3.10. Example of constraint in JML specification.

Class `ConstraintExample` in Figure 3.10 includes an example of constraint in JML. The example defines the public field `index` and the method `intcIndex`. The constraint `index == \old(index + 1)` states that after execution of the method incIndex (in a post-state), the specified constraint must be checked. If the constraint does not hold, an error is raised indicating the constraint assertion violation. A scope problem is observed in the constraint defined in Figure 2.15. Here, the constraint is applied to all methods in class `ConstraintExample`. In this way, we can have calls to other methods in class `ConstraintExample` which do not modify the field `index`, thus leading to a constraint assertion violation. To deal with such a scope problem, JML also allows one to specify constraints that are applicable only to a specific set of methods.

```
//@ constraint index == \old(index + 1) for incIndex();
```

Below we show how to rewrite the constraint in order for it to be obligatory only for method `incIndex()`. If the for clause is omitted, it becomes an universal constraint that constrains all methods. The constraint specification in Figure 3.10 is an example of universal constraint.

As with invariants, JML makes a distinction between instance constraints and static constraints. An instance constraint must hold only for instance methods, whereas a static constraint must hold both for instance and static methods. It is important to note that: (1) instance constraints do not apply to constructors and finalizers because there is no well-defined pre-state for constructors and no post-state for finalizers. Moreover, helper methods, like invariants, do not have to preserve history constraints; and (2) unlike instance constraints, static constraints must be satisfied by all constructors (after constructor execution).

Concerning constraints and method termination, just like invariants, constrained methods must preserve constraints in the case of normal and abnormal terminations. In relation to access modifiers, like invariants, constraints can also be declared with any Java's access modifiers (`private`, `protected`, and `public`). The rules of visibility for invariants are the same on constraints.

Specification for interfaces and inheritance

As we previously mentioned, JML is used to specify Java's modules such as classes and interfaces. Thus, interfaces may have specifications as well. The JML specifications that are present into interfaces provide two semantics differences between JML and Java:

- stateful interfaces — in Java, interfaces are stateless in the sense that there are no time-varying fields in interfaces. In JML, however, interfaces become stateful as one can declare instance fields into interfaces by using model fields (this is one feature of the model programs). Accordingly to JML, locations should be allocated somewhere for storing state information attributed to the interfaces;
- multiple inheritance — Java allows only single inheritance of code whereas JML supports multiple inheritance of specifications. That is, an interface in JML can have its own (model) fields and (model) methods.

In JML, a subclass inherits specifications such as preconditions, postconditions, and invariants from its superclasses and interfaces that it implements. An interface also inherits specifications of the interfaces that it extends. The semantics of specification inheritance resembles that of code inheritance in Java; e.g., a program variable appearing in assertions is statically resolved, and an instance method call is dynamically dispatched (30).

There are several ways to inherit specifications: *subclassing*, *interface extension* and *implementation*, and *refinement*. A subtype inherits not only fields and methods from its supertypes, but also specifications such as pre- and postconditions, and invariants. To provide the effect of specification inheritance, JML employs the keyword `also`, which denotes a combination (join) of specification cases, which consist of clauses including pre-, postconditions and so forth.

3.1.4. **Visibility of Specifications**

As in Java, JML provides rules of visibility for JML annotations. It imposes extra rules to the usual Java visibility rules. One rule states that an annotation cannot refer to names (e.g., fields) that are more hidden than the annotation visibility. Suppose *x* is a name declared in package *P* with type *T (P.T)*, by applying this JML visibility rule, we have the following restrictions:

- An expression in a public method specification can refer to *x* only if *x* is declared as public;
- An expression in a protected method specification can refer to *x* only if *x* is declared with public or protected visibility, and *x* must also respect Java's visibility rules — if *x* has protected visibility, then the reference must occur from within *P* or outside *P* only if the reference occurs in a subclass of *T*;
- An expression in a method with default (package) visibility can refer to *x* only if *x* is declared with public, protected or default visibility, and *x* must also respect Java's visibility rules — if *x* has default visibility, then the reference must occur from within *P*;

Note that the visibility access modifiers used in JML only occur in heavyweight method specifications. In a lightweight method specification, the privacy level is the same of the method. For example, a public method with a lightweight method specification is considered to have a public visibility annotation.

Still regarding privacy of specifications, JML also offers the keywords `spec_public` and `spec_protected`. Both are used to provide means to make a declaration that has a narrower visibility become wider (public or protected), and thus, respecting the rules of JML visibility. For instance, the declaration:

```
private /*@ spec_public@ */String name;
```

introduces a field *name* of type string that Java considers private but JML considers public. In this way, this declaration can be referred to, for example, in a public method specification.

3.1.5. **Advanced JML features**

When JML is used to specify high level properties and then to verify formally modules of programs written in modern object-oriented languages like Java, features listed in the previous section are unfortunately not sufficient. In this section we will present the necessary extra constructs for specifying object-oriented modules: datagroups, ghost fields, model fields etc.

Datagroups

It was previously mentioned that *frame properties* specify which parts of the system may change as a result of a method execution. So, any location left outside the frame property (expressed by the assignable keyword) is guaranteed to have the same value as in the pre-state.

Consider the following example from Figure 3.11.

```
public class Time {
  // some invariant
  protected int hour, min;

  . . .

  /*@ requires ...; // some pre-condition
    @ assignable hour, min;
    @ ensures ...; // some post-condition
    @*/
  public void setTime(int hr, int m) {
    this.hour = hr; this.min = m;
  }
}
```

Figure 3.11. Frame properties.

In the Figure 3.11 we have the class Time with two variables hour and min. Their values get updated by the setTime method. Note that the assignable clause of this form is highly problematic because it contradicts good object-oriented practices for two reasons: 1) it exposes implementation details by mentioning names of the protected fields and 2) the specification is restrictive for any future subclasses.

Thus, if in the future we are required to express time more precisely (e.g by adding a field for seconds and milliseconds respectively) then the newly added fields in the extended class would not be explicitly mentioned in the assignable clause.

Datagroups (31) provide a solution to this problem. The idea is that that a datagroup is an abstract piece of an object's state that may still be extended by future subclasses.

The specification in Figure 3.12 declares a (public) datagroup _time and declares that the three (private) fields belong to this datagroup. This avoids exposing any private implementation details, and it's subclasses may extend the datagroup with additional fields.

Using the datagroup principle the example from Figure 3.11 is modified as presented in Figure 3.12.

```
public class Time {
  //@ public model JMLDataGroup _time;

  protected int hour;     //@ in _time;
  protected int min;      //@ in _time;
  protected int sec;      //@ in _time;
  protected int milisec;  //@ in _time;

  . . .

  // the rest of the class
  }
}
```

Figure 3.12. Datagroup with three fields.

Model fields

Model fields serve for the purpose of data abstraction. A model field is used for specification-only purposes that provide an abstraction of (part of) the concrete state of an object. The specification in Figure 3.13 illustrates the use of a model field.

Note that the `_currentDay` model field in class `Month` is public, and hence visible to clients, though its representation is not. The `represents` clause must be declared private, because it refers to private fields. For every model field there is an associated datagroup, so that the model field can also be used in `assignable` clauses. An object can have several model fields, providing abstractions of different aspects of the object (31).

```
public class Month {
   //@ public model long _currentDay;
   //@ private represents _currentDay = month + date + day;

   private int month;   //@ in _currentDay;
   private int date;    //@ in _currentDay;
   private int day;     //@ in _currentDay;

    . . .  // the rest of the class

}
```

Figure 3.13. Model field with its associate datagroup.

Model fields are especially useful in the specification of Java interfaces, as interfaces do not contain any concrete representation we can refer to in specifications. We can declare

model fields in a Java interface, and then every class that implements the interface can define its own `represents` clause relating this abstract field to its concrete representation.

Ghost fields

Ghost fields are also specification-only fields, which means that they cannot be referred to by Java code. They're difference with model fields is that while model fields are there to provide data-abstraction, a ghost field can provide an additional *state* of the object.

The change of the state of the ghost fields is done by the `set` keyword. The setting of the ghost fields can be (and usually is) done not in the specification, but rather during the program control. In this way the objects state can be monitored during different program execution paths.

```
public class Person {
   private String name;
   private int age;
   //@ public ghost boolean _invalidAge = false;

   //@ requires age >= 0;
   //@ assignable age;
   //@ ensures age >= 0;
   public void setAge(int age) {

     if (age < 0)  {
         age = 0;  // set age to zero, since invalid value
                   // was provided
         //@ set _invalidAge = true;
     }
      else {
            this.age = age;
            //@ set _invalidAge = false;
           }
}
```

Figure 3.14. Setting the value of the ghost field based on program execution.

In Figure 3.14. we provide an example of using ghost fields to monitor the state of object based on input provided from the user. Namely, in the method `setAge` of class `Person`, if a negative parameter is provided as input, than the value of the ghost field is set to `true` indicating that the age is invalid. Note that since the ghost field was declared public it can be seen, and thus this objects state verified, from outside the `Person` object itself.

3.1.6. **Some additional JML operators**

Java Modeling Language has some operators which are useful for the definition of the method API's. The operators mentioned here can be used in both normal and exceptional preconditions (i.e. in `ensures` and `signals` clauses) and in history constraints.

The first mentioned, is the `\not_assigned` operator which specifies the data groups that were not assigned during the execution of the method being specified (29). The example below specifies that the datagroups `_gr1` and `_gr2` defined in the class `Person`, are not assigned during the execution of the method `increaseWeight`.

```
public class Person {
   . . .
   int weight;

   //@ public model JMLDataGroup _age, _stamina;

   /*@ requires  true;
     @
     @ ensures      (\result = \old(weight) + a) &&
     @               \not_assigned(_age, _stamina);
     @*/
   public int increaseWeight(int a) {
         weight =  weight + a;
   return weight ;
   }

}
```

Figure 3.15. Restricting field modification with the `\not_assigned` operator.

The next operator is the `\not_modified` operator used to assert fields whose values are the same in the post-state as in the pre-state. For example, the `\not_modified(xval, yval)` says that `xval` and `yval` have the same value in the pre and post state (their values are equal) of the method execution (see Section 11.4.4 in (29)).

The operator `\only_accessed` specifies that the method's execution only reads from a subset of data groups named by the given fields. For example, `\only_accessed(xval, yval)` says that no fields of the data groups named `xval` and `yval` were read by the method. This includes both direct reads in the body of the method, and reads during calls that were made by the method (and methods that this method might have called, etc.) (see Section 11.4.5 in (29) for more).

The operator `\only_assigned` specifies that the method's execution only assigns new values to a subset of the data groups named by the given fields. For example, `\only_assigned(xval,yval)` says that *no* fields, outside of the data groups of `xval` and `yval` were assigned by the method (see Section 11.4.6 in (29)).

Keyword	Description	return type
\not_assigned	specifies the data groups that are not assigned during method execution	boolean
\not_modified	specifies fields whose values should remain the same in the post-state as in the pre-state.	boolean
\only_accessed	specifies that only a subset of data groups named in the given fields should be read during the method execution	boolean
\only_assigned	during the method execution only a subset of the data groups are assigned new values	boolean
\only_called	method's execution only called from a subset of methods given in the method-name-list	boolean
\only_captured	method's execution only captured references from a subset of the data groups named by the given fields	boolean
\fresh	objects were freshly allocatedbound as a result of this method execution. They did not have values assigned in the pre-state.	boolean
assert	check that the specified predicate is true at the given point in the program	
assume	assume that the given predicate is true	

Table 3.2. The description of some of the JML predicates and specification expressions.

The `\only_called` operator specifies that the method's execution is only allowed to call from a subset of methods given in the method-name-list. For example, `\only_called(p,q)` says that no methods, apart from `p` and `q`, were called during this method's execution.

Similarly, the operator `\only_captured` specifies that only the method's execution only captured references from a subset of the data groups named by the given fields.

Another handy operator is the `\fresh` operator which specifies that the objects created in this method are freshlly allocated; for example, `\fresh(x,y)` asserts that `x` and `y` are not null and that the objects bound to these identifiers were not allocated in the pre-state.

Two other keywords of interest are defined here. Namely, the `assert` and `assume` statements. An assert statements tells JML to check that the specified predicate is true at the given point in the program (29). The runtime assertion checker checks such assertions during execution of the program, when control reaches the assert statement. The `assume` statement is used when we to tell the analysis tool that the given predicate is assumed to be true (29).

The difference between the assert and the assume statement is that that the static checker just uses the assumption without checking if it is correct, while with an assert statement, it checks if a condition can be deduced.

3.1.7. **Limitations of JML**

Despite that JML is able to specify quite rigorous specification it has limitations which have a particularly impact when it comes to specifying properties related to security.

JML specifications are *restricted to a single class*. This makes it especially hard to reason about security related behavior because some of the design decisions determining this behavior are made in different classes. Furthermore, at least in larger (Java) desktop applications, this behavior is usually spread in many classes located in different packages.

It is difficult to specify the allowed sequences of method calls with JML. Such sequences are essential properties of reusable object-oriented classes and application frameworks.

Another key limitation is related to one of the design goals of JML. JML was designed for specifications exclusively *restricted to the functional behavior* of programs. For this reason most of the up to date security related research involving JML tends to reason about security from a functional point of view as well. This means that reasoning about security properties with JML depends from the implementation of solutions for those security properties. JML as a *design by contract* language, in this context is used to reason about the correctness of implementation of these solutions.

The third limitation of JML is that it has *not much support for refinement*. The only refinement support that JML has is the possibility to express the data abstraction of fields of objects by the JML construct of model fields [For more on model fields go to page 28]. No refinement is possible for any JML clauses representing behavior. As a result of this, it is not possible to define any sort of dynamic annotations, that would for example reflect on behavior based on a given pre condition.

The motivation for JML is to provide support for ensuring that the program works under the assumptions the programmer had in mind during development (32). That is, improve program development so that assumptions of specific actions are recorded and enforced. The specific actions ensure that method implementations behave as expected.

3.1.8. **A comparison of JML and OCL**

JML does have some things in common with the Object Constraint Language (OCL), which is part of the UML standard. Like JML, OCL can be used to specify invariants and pre- and postconditions. An important difference is that JML explicitly targets Java, whereas OCL is not specific to any one programming language.

JML clearly has the disadvantage that it can not be used for, say, C++ programs, whereas OCL can. But it also has obvious advantages when it comes to syntax, semantics, and expressivity. Because JML sticks to the Java syntax and typing rules, a typical Java programmer will prefer JML notation (33) over OCL notation.

JML supports all the Java modifiers such as `static`, `private`, `public`, etc., and these can be used to record detailed design decisions. Furthermore, there are legal Java expressions that can be used in JML specifications but that cannot be expressed in OCL.

More significant than these limitations, or differences in syntax, are differences in semantics (26). JML builds on the (well defined) semantics of Java. So, for instance, the `equals`

keyword has the same meaning in JML and Java, as does ==, and the same rules for overriding, overloading, and hiding apply. This is not the case for OCL.

3.2. Java RMI

In this section we present Java Remote method Invocation mechanism that we used during the implementation of the prototype in our case study. Namely, first we give a short overview of the Java RMI technology (that has been in existence for more than twenty years) and secondly, we give an outline of necessary steps to create distributed objects with this technology.

3.2.1. Remote Method Invocation (RMI)

The primary goal of Remote Method Invocation (RMI) is to make it possible to develop distributed Java applications with same syntax and semantics as if they were used in non-distributed applications. RMI allows developers to create programs that *share objects* (usually located on the server side) and make them remotely accessible for other programs (usually on client side), allowing them to invoke methods and access attributes in the same way as it is done in local objects. RMI requires only a TCP/IP based network infrastructure for communication between Java virtual machines, but all network related communication is hidden from the developer.

The core of the RMI lays in the *definition of objects* and the *implementation of objects* (34). In RMI, the definition of a remote object is coded using an interface. The implementation of the remote object is coded using a class. Clients that invoke methods on the remote object use the interface (definition) of the remote object and do not know the implementation.

Figure 3.16 shows an example of remote method invocation. Both the client and server know the `Example` interface. The server has the implementation of this object and has made it available to the client to contact it remotely. The client only knows the definition of the class; methods of this class that are invoked on the client side will function as a proxy.

The way the server shares the objects is by placing them on the *RMI registry* using a string as a name to identify them (34). When clients want to access the objects that the server advertises, they request the *name* together with a *hostname (Internet address)* to identify the virtual machine that serves the object.

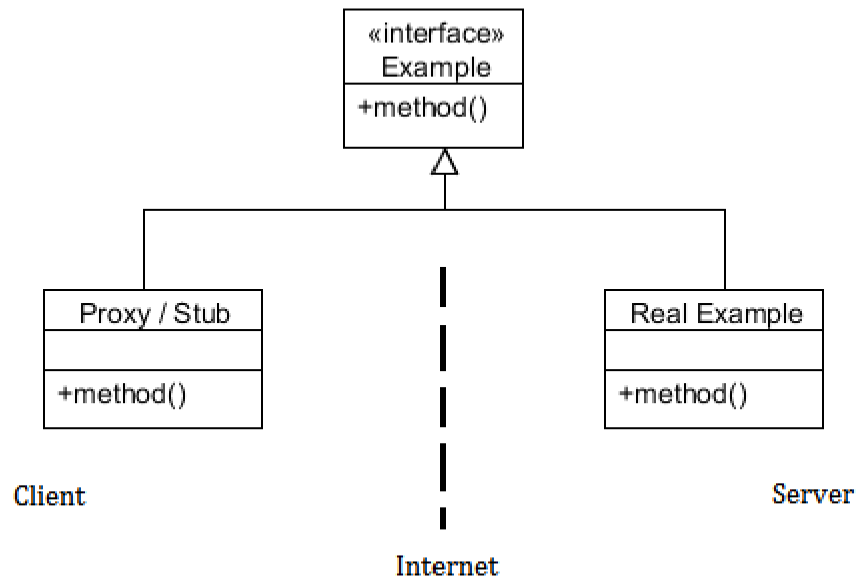

Figure 3.16. The conceptual overview of distributed objects with RMI.

Steps for creating distributed objects with RMI

Below we show steps in a typical usage of RMI technology. The steps are organized as follows:

1. Creating an interface

```
import java.rmi.*;
public interface ServerObj extends Remote
{
    public void methodA() throws RemoteExcetption;
}
```

This is the interface of the `ServerObj`. By extending the `java.rmi.Remote` class, the objects that implement this interface can be available remotely. The method signatures need to throw `RemoteExcetption` exception because they are not executed locally.

2. Creating the implementation

```
public class ServerObjImpl extends java.rmi.server.UnicastRemoteObject
{
   public void methodA()throws java.rmi.RemoteExcetption
   {
     // … implementation code
   }
}
```

The `ServerObjImpl` is the implementation of the `ServerObj` class. It extends the `UnicastRemoteObject` to have it registered into the RMI registry and thus make it available to get called remotely.

3. Serializing objects

In order to be able to pass objects as arguments to remote methods, *object serialization* must take place. Therefore each objects that will be passed through the network has to implement the `Serializable` interface.

This way, these objects can be flattened and streamed over the network.

4. Creation of Stubs and Skeletons

In Java, manual creation of the stubs was necessary until the version 1.4.2 which coincidently, is the version that we will be using.

To create the stub and skeleton files, the RMI compiler is used. It is a command line tool that is shipped with the Java SDK and is named `rmic`. After running it on `ServerObjImpl.class` a `ServerObjImpl_Stub.class` and `ServerObjImpl_Skel.class` are created.

In these two files resides all the network communication details. So when the client calls a remote method he is actually calling the stub will contain the code to connect to the remote skeleton, to send details about the function to be invoked, to send the parameters, and to get the results back.

5. The server side

```
ServerObj s = new ServerObjImpl();
Naming.rebind("rmi://localhost:1099/Service”, s);
```

Naming.rebind() is a static method that is used to bind a remote object to an URL. The number 1099 is the default port that is used with RMI registry.

6. Client Side

```
ServerObj so = (ServerObj);
Naming.lookup("rmi:// localhost:1099/Service");
```

`Naming.lookup()` returns a `java.rmi.Remote` object (proxy stub), that can be used like a normal, local object.

7. Running the Applications

First, the RMI registry server has to be started where we register the remote object `ServerObj`. Next, the client can be started. It will connect to the registry and obtain a reference to the `ServerObj` by specifying the name that was used by the server to register it.

3.3. Authentication and Access Control in Java

In this section we present Java Authentication and Authorization Service (JAAS). We use this mechanism to implement security properties in our prototype application.

JAAS was introduced in the Java 2 SDK. It represents a framework that authenticates users and allows them (or not) to run certain code, based on the privileges that were assigned previously for each user individually (35).

3.3.1. Java Authentication and Authorization Service

JAAS is mainly formed of two components, *authentication* and *authorization*. The authentication part is used to determine the identity of the user, while the authorization component is there to restrict the execution of the sensitive tasks that users can perform. The latter component does this by checking if the (authenticated) user has privileges to run that particular source code.

JAAS is a *pluggable* interface that can be developed independent of the application and irrespective if it's an applet, servlet or an ordinary desktop application. (35)

Subjects and Principals

Users or processes in JAAS are represented as *Subjects* which represent *single identities* and are implemented by the `Subject` class (36). A single subject class can have numerous associated identities, each of which is represented by a `Principal` object.

For e.g., a single `Subject` represents a medical employee which requires access to both, email system and the patient database. This `Subject` will have two `Principals`, one associated with the employee's user ID for e-mail access and the other associated with his user ID for patient database. Each time this user logs in successfully, both of these principals are (dy-

namically) associated to him gaining him execution privileges for both of these functions. When the user logs out, all his associated principals are removed from the subject.

Policy files

Policy files are the main mechanism to control access to system resources and sensitive code execution. They can be used for three main purposes, to specify the login configuration type, to assign system resources and to assign execution privileges to authenticated users.

```
grant signedBy "Erik", codebase "file:Querry.jar"
{
    permission java.io.FilePermission "/DB/Access", "read";
}
```

Figure 3.17. The policy file `my.policy` for access control.

Access control to system resources is specified in a separate file and loaded at start-up as an argument. This file contains statements as shown in the Figure 3.17. This policy file serves to create the `java.security.policy` object.

Statements in Figure 3.17 allow code signed by "`Erik`" located in the file `Querry.jar` file *read* access privileged to the files in `/DB/Access` directory.

In this way, `Subjects` discussed previously, are assigned specific system resource permissions through the policy file.

To authorize code execution-privileges to certain `Principal` of a `Subject`, we can give a policy as an argument to the `java.security.auth.policy` object with `grant` entries that look as in the Figure 3.18.

```
grant Principal MyPrincipalIml "user1"
{
     permission PersonnelPermission "*";
};

grant Principal MyPrincipalIml "user2"
{
     permission PersonnelPermission "doOperationA";
};
```

Figure 3.18. Assigning different privileges to different users.

In Figure 3.18 `user1` is permitted to execute all methods, while `user2` is permitted to execute only the method named `doOperationA`.

Every time we try to execute code whose access is monitored by JAAS in that access control context, a facility called `AccessController`, iterates through the permission objects

created by the `grant` statements that were listed in the policy file for that particular `Subject` (35).

Login Context

To set up a login module for an application we need to implement the `LoginContext` class.

The `LoginContext` class authenticates `Subjects` by using built-in methods. This class can take two parameters, the `Configuration` object that associates the login module to be used in the application and a `CallbackHandler` class. The `CallbackHandler` reads the username and password[9] from the user.

3.3.2. Flow of operation in JAAS

As we mentioned previously, in JAAS operations are split in two phases. First, a Subject has to be authenticated and then various JAAS authorization features can be used to restrict access to the system resources. Both phases of authentication and authorization are explained below.

User authentication in JAAS

JAAS establishes the identity of the user based on the credentials the user provides. Figure 3.19 presents the sequence diagram of processes when JAAS establishes the identity of the user.

To authenticate a `Subject` instance, an application instantiates a `LoginContext` object. This is done so the credential information of the `Principals` are presented to JAAS. The `LoginContext` instance loads the login module(s) that was configured for that application. Then, the application invokes the `login` method of the `LoginContext` class. The `login` method, in turn, invokes all the login modules, which call the `CallbackHandler` to read the *username* and *password* from the user represented by a `Subject`. If these modules successfully authenticate the `Subject`, they (according the authorization policy `grant` entries supplied) *associate* his corresponding `Principals` with that `Subject`. The `LoginContext` class returns the (authentication) status to the application. If the authentication is successful, the user's identity has been verified and this `Subject` instance is retrieved from the `LoginContext` instance.

Now that the `Subject` has been authenticated, a fine-grained access control can be enforced to verify his permissions before the JAAS systems allows him to perform actions that he requests.

[9] Note that different features, such as voice recognition, can be used to determine the identity of the user.

Verifying the authorizations of users with JAAS

After the identity has been established and the `Subject` class created for that identity, a fine-grained access control can be enforced by invoking the built-in `doAs` and `doPrivileged` methods of this class. Using JAAS access control can be enforced every time when user acesses/creates different system resources such as files on the disk, different system options etc.

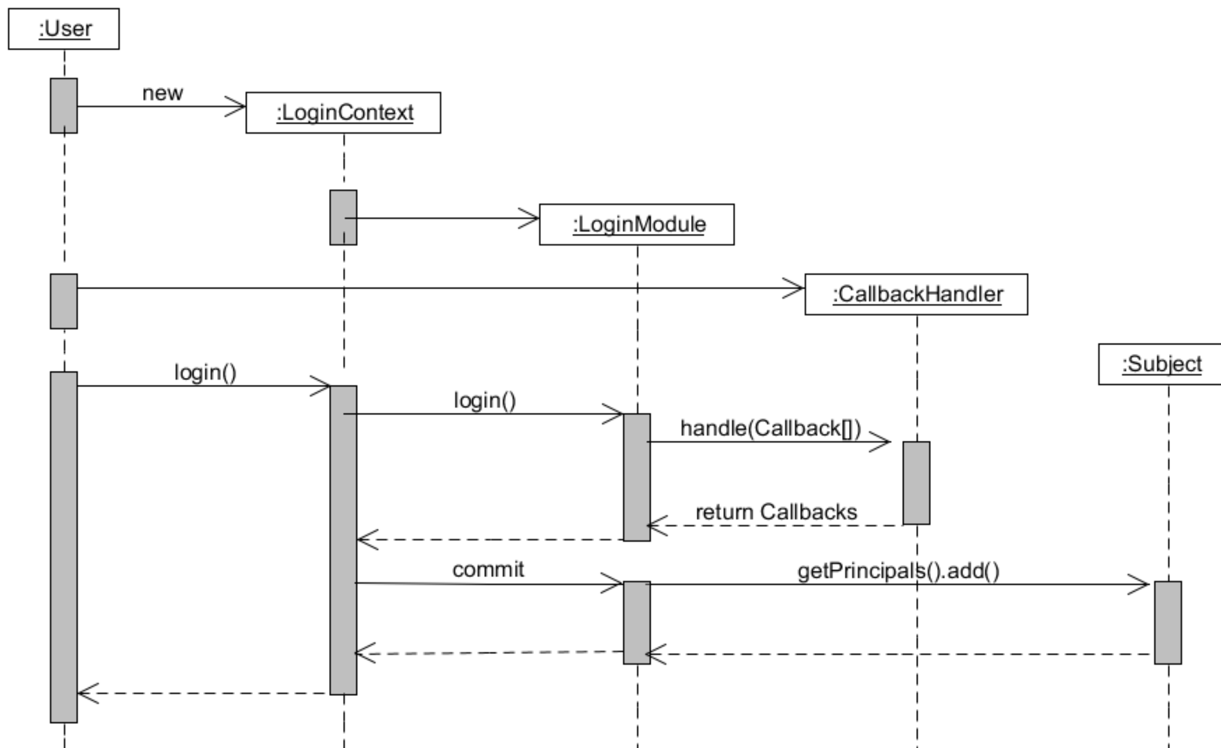

Figure 3.19. A typical JAAS Authentication sequence.

Related to authorization control, in our application we mainly use JAAS to ensure that only users who have permissions for the methods that they attempt to call are actually allowed to do call them. To provide access control of methods we use the built-in methods of `Subject`[10] and `AccessController`[11] class of Java API.

By invoking the internal `doAs` method the `Subject` class, verifies if it a permission object named like the method he is trying to call. To verify that the call done by the `Subject` is in compliance with the current security policy the `checkPermission` method of `AccessControl`

[10] javax.security.auth.Subject
[11] java.security.AccessController

is called. If this call is allowed, then `checkPermission` returns quietly. If the call is not permitted, an `AccessControlException` is thrown.

This concludes the control of authorization for that `Subject` making the method call.

After discussing the background concepts and programming technologies that will be used during this thesis work, we continue in the next chapter, by looking at the possible approaches where JML was used to specify programs.

Chapter IV

4. Approaches of using JML to specify security properties

In this chapter we discuss the approaches that we have identified on how JML was used before to reason about security properties. Some approaches attempt to *formally* verify the properties or while others just to prove that their program is *correct* in relation to their intent. We present examples of these approaches and explain the approach that we plan to use when specifying security properties.

One approach views security properties as a problem of correct implementation of the intended behavior. The problem here is how to ensure that the functional part of the program which implements the security properties is implemented correctly. *Correctly* in this context means that the implemented functionalities match the behavior that the developer had in mind. The goal of this approach is to verify that the behavior which was specified with JML (as the intended behavior), really matches the actual implementation. And if so, it is claimed that because the implementation is correctly implemented, the specified properties also hold for that implementation.

Another approach identifies states in the program and models the program, based on the states, as a finite state automaton. The security related behavior of the program is also modeled in the state automata. The problem here is how to capture the behavior and state transitions of the program with JML constraints. The goal is to verify that transitions of states in the program match the transitions in the state automata. If this is verified then the program meets the security properties modeled in the automata.

In the field of language based security, the security properties of confidentiality and integrity are defined in terms of non-interference in the data flow. The fields involved in the data flow are assigned different security levels. The problem here is how to ensure that confidential data from fields belonging to the higher security levels don't leak to the lower fields intended for public data. The goal is by using JML, how to specify that *no* data flow should occur from high (secret) fields to low (public) fields.

We will explain these approaches in more detail below. During our explanations, on occasion, we use the following abbreviations:

Abbreviation	Represents
P	The program that implements the intended behavior (where the JML specifications will be inserted)
S	The down finite state automata modeling the states of the program P
SJ	All the JML statements used to represent S
Φ	The property that we are specifying

Table 4.1. Explanation of used abbreviations.

Each of the following three sections address one the afore mentioned approaches. Each section illustrates the approach with example(s). At the end of each section we identify issues related to that particular approach.

4.1. Correctness of implementation

This approach of specifying security properties views security from the *correctness* point of view. Some of the examples of this approach are (14), (37) and partly (38).The foundation of this approach lays on the principle that even though a correct implementation does not necessarily represent a secure application, the opposite of this claim certainly implies (among others) an erroneous and possibly insecure application. Thus, it is considered important to have a *correct implementation* of the intended behavior. In this sense JML is used as a second assurance for the correctness of implementation. Thus, we have the code behavior establishing the given property, and we have JML statements which should capture the requirements of the method implementor and responsibilities of the method caller.

This approach starts by claiming that the program P meets certain security properties because of its specific implementation details. It informally argues that because P has certain features it meets the claimed properties. An example claim can be, since P implements a cryptographic protocol when party A communicates with party B, this program meets the property of confidentiality of communication between these parties.

However, to be sure that the implementation of P really implements the argued behavior *correctly* it uses different JML constructs to capture the behavior of P in annotations. In other words, in order to be sure that confidentiality is achieved we specify the behavior of the code implementing the cryptographic protocol used in communication between party A and party B. Inserted JML annotations are to be understood as predicates for the code (17).

We denote the total of JML statements required to annotate P as SJ. Thus, SJ would for example specify the behavior of code in terms of when a nonce is generated, how public and private keys are being administered, how the program encrypts and decrypts the provided messages etc.

Further on, static analysis tools (such as ESC/Java2) [12] are used to verify that `P` reports no violations in relation to the `SJ`. If no violations are reported then by inference it claims that `P` has been implemented correctly in relation to `SJ` and thus it meets the claimed property.

In essence JML here is used to specify the behavior of the code that has been implemented. The motivation for this is twofold, first; some of the basic errors can be reported by static analysis tools such as lack of null checks, unnecessary leakage of object references etc. and secondly, by reasoning about the code in the contractual manner between the caller and the implementor, it should contribute to a better understanding of the code and point out possible mistakes or design errors made during the implementation phase (26).

In the following, we provide a simple example to show how JML is used to verify properties with this approach. Note that this program is a simple "toy" example and is showed hereby to only illustrate the principle of how JML can be used to claim that an implementation of a program is *correct* and thus that program meets the given property.

An example

Our simple example is presented to only show is this principle used to claim correctness of the application. The program in question `P` presented in Figure 4.1 is very simple and has one policy defining confidentiality. The policy specifies that the program only prints to the screen objects of type `UserData` that are *not* confidential and ignores the received confidential objects of the same type. In this example we show how this behavior respecting the given policy can be modeled by JML. The purpose of the modeling is that, if the annotated P is statically verified then we can claim that the policy is correctly implemented and thus P is confidential in respect to the definition of confidentiality in our policy. Note that usually security properties are more complex and programs larger, thus the JML constructs used to specify those properties are more advanced than the ones used in this example. However this example should suffice to illustrate the principle.

The method starts by claiming that the java program `P` presented in Figure 4.1 has the property Φ. `P` contains a class `UserData` with two fields to hold a username and password as well as a `boolean` field named `isConfidential` to keep track of state of that object during program execution. The `isConfidential` field is updated during the program flow at different points according the program logic that we are implementing.

Below we show only the relevant parts of P and hide the rest for clarity. The other elements such as the definition of `PrintException` and `toString` method are omitted in the Figure 4.1 as they do not contribute the purpose of our explanation.

The definition of property Φ which should hold in `P`, states that *the method* `printToScreen` *prints only objects of type* `UserData` *which are not marked as confidential and ignores the rest.*

[12] Available at: http://secure.ucd.ie/products/opensource/ESCJava2/

```
class P {

   class UserData {
        private String username;
        private String password;
        private boolean isConfidential;

        public UserData(String usr, String pwd) {
             username = usr;
             password = pwd;
        }
        . . .
   }

   . . .

   public boolean printToScreen (UserData ud)
                                 throws PrintException {

        if (ud.isConfidential==true) {
             throw pe;
        }
             else {
                  ud.toString();
                  return true;
             }
   }

   . . .

}
```

Figure 4.1. Program `P`.

Our claim that property Φ holds in program `P` is based on the particular behavior that we have implemented in `P`. This behavior is defined as follows: "the method `printToScreen` in program `P` checks if the parameter object received has the field `isConfidential` set to `false`. If so, it prints his string representation to screen and if not, it throws a `PrintException` without printing the object". Based on this behavior we claim that `P` fulfils property Φ. The behavior explained in the statement below is implemented in the `if` statement of the `printToScreen` method of program `P`.

However, in order for us to be sure that `P` really implements Φ we use JML to express the intended behavior with annotations. After annotating the program with JML it now becomes:

```
class P {

   . . .
   class UserData {
        private /*@ spec_public @*/ String username;
        private /*@ spec_public @*/ String password;
        private /*@ spec_public @*/ boolean isConfidential;
   }

   . . .
        /*@ requires usr!=null && pwd!=null;
          @ ensures  username==usr && password==pwd;
          @*/
        public UserData(String usr, String pwd) {
             username = usr;
             password = pwd;
        }
        . . .
   }

   . . .

   /*@ requires ud!=null;
     @ ensures  \result==true;
     @
     @ signals_only (PrintException pe)
                                 ud.isConfidential==true;
     @*/
   public boolean printToScreen (UserData ud)
                                 throws PrintException {

        if (ud.isConfidential==true) {
             throw pe;
        }
             else {
                  ud.toString();
                  return true;
             }
   }
   . . .

}
```

Figure 4.2. Annotated version of program `P`.

By inserting the JML annotation we capture the previously defined behavior of `P` with JML. Note that we have additionally required a check of the `isConfidential` field by JML be-

fore we print the object. This should make it clearer that such check is necessary in the program code as well. By doing the annotations of our program the requirements should become clearer leading to a better understanding of the code.

This annotation ensures that when the `printToScreen` method is called, the object received as parameter should not be null and is required to have the `isConfidential` field set to `false`. Only if this holds, the object will be printed, on the contrary a `PrintException` is raised.

So, when a static analysis tool is used to verify `P` against the provided JML annotations and if no violations are reported during verification, then our claim is considered to have been verified. In that case, program `P` does indeed meet property Φ because the behavior in `P` is implemented correctly in relation to the annotations which specify the claimed behavior for establishing property Φ - as it has been defined in the policy. Note that because the program we have shown is very simple, the JML used here is straight forward. When the program is more complex and when policies deal with more advanced security properties, then these JML constructs do not suffice any longer, new and more advanced ones have to be used.

Despite that the JML features used with this approach depend on the specific program behavior being implemented and the properties that are being specified, there is a group of constructs from JML that are often used when specifying properties with emphasis to security. Next we present the most important constructs of this group.

4.1.1. Employed JML constructs

The JML constructs used most often for preventing coding errors that might lead to a compromised security in the application are listed below. They include a special specification-only field marking the state, object invariants keeping track of consistency through the object life cycle, pre- and post- conditions expressing context of method calls and exception control.

Ghost fields

Ghost fields are used to mark the states when analyzing the information flow property (39) of data. Variables of this kind are auxiliary fields which are not used by the implementation, but occur in specifications only. For example if we have an object that is being received by another object and if we mark the state of the first object with a ghost variable like below:

```
//@ public ghost boolean isConfidential = false;
```

In the precondition of the method that receives this object we can ensure that only the objects with the appropriate state are received. Furthermore, we can specify this as an invariant in the receiving object, at which case we ensure that the received object maintains one of the states from a fixed set of previously defined states. This enables us to specify the state of that object during its flow through the program.

Information flow is just one example where ghost fields can be used. Besides using them for information flow monitoring they can be used to mark any state in the code by only setting their value appropriately at that point in the code.

Object Invariants

Object Invariants express properties which should hold at the entry and exit to each method. The invariants describe the meaning of the consistency of the object data.

An object invariant that verifies if the state of the receiving object is true looks as in the following example:

```
//@ invariant obj.isConfidential == false;
```

Here `obj` is the reference of the object being received. In this way, when objects are received we can check if they are in the appropriate state. In the information flow control domain, this allows us to monitor the flow of data in the program based on their current state. Note that is possible only for immutable types as ghost fields can only be assigned to them.

Pre- and Post- Conditions

Each method in Java code is supposed to be called in certain context i.e. it assumes that certain fields of its object are appropriately set, that the parameters come from specific ranges (e.g. between 0 and 10), that a particular parameter has a particular type, and in general that certain relations hold between the input data and/or the fields of the object. Here is an example of such a precondition:

```
/*@ requires obj.isConfidential!=true && param1!=null
  @*/
private String m(Password obj) {
  . . .
}
```

Figure 4.3. A precondition for method `m`.

The above invariant states that when method m is called it should only accept `Password` objects of type `confidential`, that is, where the state variable named `isConfidential` is set to true.

Similarly, it is often needed for a method to guarantee that certain fields are set, certain relations between the object states hold or, a relation between the result and the input data of the method hold. This is done by means of `pre`– and `post`– conditions.

```
/*@ … ensures \result.isConfidential && \fresh(\result) …
  @*/
private … String m(Password obj) {
 . . .
}
```

Figure 4.4. A postcondition for method m.

For example if we want to specify that the result of m is confidential and should be protected from exposure in the code of the application we annotate as in Figure 4.5.

Additionally we have required that the result generated after the method m is executed needs to be generated inside this method. This solution is one way how to prevent an uncontrolled aliasing of data considered to be confidential.

Exception control

The specification of exceptions can be used to clearly mark the conditions guaranteed to hold after an exception has been raised. This leads to a clearer understanding the code. It can also mark types of exceptions that are possible to be thrown and the pre- and post- conditions separately for each exception.

```
/*@ … signals_only (NullpointerException e) param1==null  &&
  @ …   signals_only (SecurityException se)
  @*/ obj.isConfidential==false
private … String m(String param1, Password obj) {
 . . .
}
```

Figure 4.5. Catching the behavior of exceptions in JML.

In this invariant we have specified that when method m is called, if the provided param1 is null or if the object obj of type Password is not confidential then the corresponding exception will be raised.

4.1.2. **Issues**

We have stated previously that this method of specifying security properties enables us to capture the *intended* program behavior by JML annotations and then verify that the code correctly implements that behavior. And since this is the case then we claim that security properties claimed in the implementation are also met.

However, in principle there is no guarantee that the annotations are really correct, or that they really capture the intended behavior accurately. As (14) claims; the annotations do

not provide full proof security but only an additional standardized level of assurance that the source code is well written with regard to the intended behavior.

This approach brings to a better clarification of the requirements of the software, because reading of the specifications is easier than reading the actual source code as the annotations provide a certain abstraction of the functionality (14). They also give a stable representation of the expected functionality during development which can be a huge asset in areas of development where cost of implementing flaws is way too high. This cost can be matched with the higher cost during development process, because adding these annotations is a costly and labor intensive process which contributes to the rise of the total cost of development.

The cost of development rises because (a) the code needs to be specified in detail with annotations and (b) the behavior of Java API's need to be specified as well in order for the verification to be possible. There have been projects like Daikon (40) to automatically detect annotations. However these annotations do not represent the intended behavior but the observed properties of the program. Therefore if the behavior of the code is not the intended one then the generated annotations are not correct either.

The explained method has been used in somewhat larger Java (desktop) applications and applets to ensure that functional and security requirements are properly implemented. Another approach aiming towards narrowing the gap between the actual code behavior and the JML annotations is the approach presented below. Namely, capturing the behavior of the program by representing it as a state machine.

4.2. **State machine representation**

This approach is widely used in the Java smart Card applications (9) and will be explained below in more detail. A similar approach is used in (16) with a slight difference that no state automata is created but the state of the program is still established and maintained by JML constraints, the same as in (9). Java applets running on browsers can also use a similar approach (41). The common ground of all these examples is that they precisely define the states occurring during program execution, express them in JML and then verify if the transitions of the states in the program match the states transitions defined in annotations.

Based on their architecture these Java Smart Card applications can easily be modeled as state machines. These applications are also called applets and usually have their *Installation*, *Personalization*, *Processing* and *Locked* states as their main life-cycle stages (42).

Each phase corresponds to a different moment in the applet's life-cycle. First, the applet is loaded on the card, and then it is properly installed and registered with the Java Card Runtime Environment. Next, the card is personalized, i.e. all information about the card owner, permissions, keys etc. are stored in the device running the programe. After this, the applet is selectable, which means that it can be repeatedly selected, executed, and deselected. However, if a serious error occurs, for example, if there have been too many attempts to verify a pin code, the card can get blocked or even become dead. From the latter state, no recovery is possible.

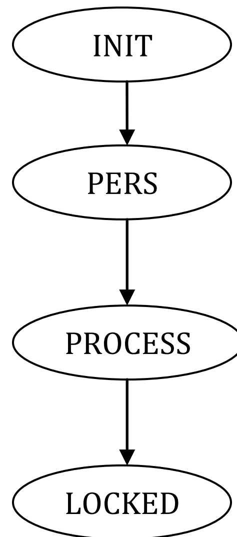

Figure 4.6. Default life cycle model for a Smart Card application.

The default stages of a smart card applet are presented in Figure 4.6. In most of the smart card applications additional stages can be derived based on the particular functionality that the smart card application implements.

This approach models all the stages of the applet in a finite state machine, which we denote as `S`. The transitions between the identified states represent the methods which transform the state of the program from one state into the other, as defined in the automaton. This change of state in `P` comes as a result of the code being executed in these methods. The methods that take the program from one state to another are specified with annotations, precisely defining their behavior. This behavior is among others, defined with pre and post conditions.

If states of `S` are modeled from states of the program `P` then we say that *S mirrors* the program P.

The part of the program behavior which ensures that security properties are met is also modeled in the finite state machine. For example, in Java Smart Card applications, modeling the following set of security properties is of particular interest: *Atomicity*, *Applet life cycle*, *Exception handling* and *Access Control* (16).

Atomicity in this context refers to the requirement that all the transactions should not be nested but atomic and, there should be no uncaught exceptions in transactions. When the applet gets into operation for the first time, this initialization should be carried out by an authenticated authority, usually by a PIN number or alike. Once it is in operation, if it gets into the blocking state, it should be able to be unblocked only by an authenticated authority, usually accomplished by a PIN number. This personalization can be carried out only once.

The program `P` closely mirrors the automata `S` because `S` was modeled based on `P`. To ensure that state transitions occurring in `P` indeed mirror the transitions occurring in `S`, JML annotations are added to `P`. The role of the annotations is to define constraints that specify legal transitions of states in the program `P`. These transitions are initiated by methods and

expressed in class constraints. The class constraints reflect the set of legal states for each object created from that class at any point in time during the program execution.

Static analysis tools can be used to verify that the order of state-changes during execution of program `P` indeed matches the one of `S`. If no violations are reported, it is claimed that `P` really matches the behavior of `S` and since `S` models the security behavior as well, it is inferred that `P` also meets the security properties of `S`.

An example

To provide an example about how JML is used to specify security properties of Java Card applets we will show the specification of *atomicity* as one of the security properties of applets. Note that in this example we will not model the whole life cycle of the applet, nor we will model the state changes in the applet execution. Our intention is to show only the specification of the property of *atomicity*. Atomicity ensures that no nested transactions are allowed to be carried out during the applet operation.

This is important in smart cards because they do not have a power supply, thus a brutal retrieval from the terminal could interrupt a computation and bring the system in an incoherent state. To avoid this, the Java Card specification recommends the use of a transaction mechanism which controls the synchronised updates of sensitive data.

A statement block surrounded by the methods `beginTransaction()` and `commitTransaction()` can be considered atomic. If something happens while executing the transaction (or if `abortTransaction()` is called), the card will roll back its internal state to the state before the transaction was started.

This transaction mechanism should ensure that (a) only one transaction happens at a time, (b) if a variable maintaining the number of transactions in place is one when the applet initializes, that means that it was previously shut down abruptly (without committing the transaction) and thus us it should revert the state back to it's previous, and (c), when the transaction is completed the variable keeping track of the numbers of transaction should go to zero enabling the applet to process the next transaction.

To enforce the proper implementation of this behavior we model it with JML by a static ghost variable named `TRANSAC`. This variable keeps track of the number of transactions in progress at any time:

```
/*@ static ghost int TRANSAC == 0; @*/
```

Each time a transaction is initiated this variable is set to one and when the transaction is finished it is set back to zero again. Thus the method `beginTransaction` is annotated as follows:

```
/*@ requires  TRANSAC == 0;
  @ assignable      TRANSAC;
  @ ensures   TRANSAC == 1;
  @ signals_only    (TransactionException te)
  @                 requires    TRANSAC == 1
  @                 ensures     te.getMessage()!=null;
  @*/
public static void beginTransaction() throws TransactionException {
   . . .
}
```

This annotation requires that there are no transactions in progress when this method is called. After it is called the value of TRANSAC should be set to one, indicating that there is a transaction in progress. And if there is a transaction already being completed when this method is called then a TransactionException is thrown. This specification also states that during the implementation of this method the value of the ghost field monitoring the number of the transactions needs to be set to one.

The method commitTransaction() has the following annotation:

```
/*@ requires  TRANSAC == 1;
  @ assignable      TRANSAC;
  @ ensures   TRANSAC == 0;
  @ signals_only    (TransactionException te)
  @                 requires    TRANSAC == 1
  @                 ensures     te.getMessage()!=null;
  @*/
public static void commitTransaction() throws TransactionException {
   . . .
}
```

The annotation of this method on the other hand, states that a transaction must have been started before commitTransaction() method can be called. Which makes sense, because there needs to be a transaction in place prior to it being committed and recorded into the smart card memory. After it is finished the state variable should be set to zero.

Note that in the annotations of this example the implementation of methods is not provided because it is straight forward.

4.2.1. Issues

This approach has mostly been applied to programs for Java smart cards[13]. There has been a considerable success in preventing errors that might lead to security flaws in these applications. Since they run on devices with very limited memory and processing capacity, the desired security properties are mostly related with this limitation of resources. For this reason these applications are small, so the state transitions in these programs can be modeled accu-

[13] http://java.sun.com/javacard/

rately and verified easily. This approach contributes to an increase of dependability of software from smart card application providers (38).

Note that applying the same approach of modeling all the program states in regular large-scale application, in comparison to applications of smart cards, can be tedious and even impossible because of one fundamental problem. Namely, the *state explosion* problem. The number of states of almost any system of interest is huge even when using different techniques of reducing this state (43).

It should be noted that this approach does also not provide full proof security for high level properties such as confidentiality, authentication or integrity. The constructed automaton which the program mirrors, is in no way verified for all possible threats against the anticipated security properties. It is there to just model the security related behavior in more detail and express it in JML so it is clearer during implementation. However after such behavior is implemented and verified we still have no guarantee that both the model and the annotations are not faulty. This is simply left on our sound judgment to decide.

Considering that smart card applications have limited complexity compared to, for example Java desktop applications, detailing their behavior in a state machine, specifying their code with JML and formally verifying it in this manner, is a strong guarantee for their security.

However, expressing program-wide high level security properties such as authentication, confidentiality or integrity is far harder to express in JML (42). This is because of the overall limitations of JML (See Section 3.1.73.1.7).

4.3. **Language Based perspective**

Another approach where JML was used to specify security properties is in the Language-Based information flow security. This branch within computer security tries to enforce end-to-end secure information flow for programs (44). In modern programming languages information flow is tackled by access control modifiers. However, this can not guarantee an end to end secure information flow. The basic question raised here is how one can ensure that public information is completely independent from sensitive (secret) information.

In (42) specification of confidentiality with JML is presented from a language based security point of view expressed from an information flow perspective. Here confidentiality is formalized using the notion of *non-interference*. i.e. it deals with all possible runs of the program that terminate normally, without for example exceptions being thrown.

The notion of non-interference was first introduced by Goguen and Meseguer (45). They define non-interference in an automata. This automata has input and output channels. The input and output channels are labeled with security levels, typically using levels `High` for secret and `Low` for public information. By fixing the low input channels and varying (all) the high input channels, one can check if high input channels are indeed independent of low outputs. In this way, we can guarantee that data flowing from the high input remains independent from the data in the low output and thus, non-interference is preserved. This work was later picked up by Volpano and Smith (46) showing that type systems are especially well suited for (automatically) enforcing non-interference properties.

In order to be able to verify that a program is non-interfering, all output and input channels need to be labeled with appropriate security levels. This labeling defines a *secure information flow policy*. Instead of just labeling the inputs and outputs of a program, typically all *global* variables (fields) are labeled and those that are only used locally are discarded.

To monitor non-interference, security levels need to be ordered in some way. We use a security lattice for this ordering and denote the ordering with the ⊆ symbol. This (basic) lattice will suffice to explain the main ideas in most cases, but if a more complicated policy is used than a more complicated lattice is required. Building such a lattice is just a matter of doing more 'book-keeping' (42). This simple Security Lattice Σ is defined below:

$$\Sigma = \{High, Low\} \qquad \text{with } Low \subseteq High$$

A secure information flow is then given by the function `Sif : Var → Σ` which maps variables to security levels in the simple security lattice. `Sif` denotes a secure information flow policy represented by a function. Security levels can now be identified as sets of variables corresponding to those levels, i.e. `High = {v ∈ Var | Sif(v) = High}` and `Low = Var \ High`.

As stated before, this approach formalizes confidentiality as a notion of non-interference. The definition of this term is taken from (45) and defined as:

> *Noninterference of programs essentially means that a variation of confidential (high) input data does not cause a variation of public (low) output data.*

From this definition we can see that confidentiality involves a relation between the pre- and post- state. In other words defining confidentiality means ensuring that we don't have a flow of data from high fields to the low fields. The reasoning behind is that data from the high fields should be kept secret from data flowing in the low fields because low fields are of lower security level than the high ones.

Integrity on the other hand involves the definition of data that keeps secret data of the high fields safe from modification coming from the low fields. It ensures that secret data flows only in high fields and there is no 'leakage' of data from high fields to the low ones.

In JML it is possible to express such relations using the keyword `\old`. If used in a postcondition, variables encapsulated by `\old` are evaluated in the precondition. This feature makes it possible to build a formulation of confidentiality in JML. Note that formalization of confidentiality is guaranteed only for executions that terminate normally (where no exceptions are raised). This non-interference is referred to as *termination-insensitive non-interference*.

With the help of non-interference this approach defines security properties of confidentiality and integrity by establishing corresponding *specification patterns* (42). The first pattern defines confidentiality as:

$$\texttt{ensures low == \textbackslash old(}\chi_{Low}\texttt{);}$$

For each `low ∈ Low` *the expression* χ_{Low} *should not contain any fields* `high ∈ High`.

A specification like this for a Java method enforces that *all* fields `low ∈ Low` in the post-state are independent of the values of fields `high ∈ High` in the pre-state. This proves confidentiality for that Java method.

The JML pattern for specification of *integrity* is:

ensures high == \old(χ_{High});

For each `high ∈ High` *the expression* χ_{High} *should not contain any fields* `low ∈ Low`*.*

The second pattern expresses that all high fields are independent of low fields. Thus low fields cannot alter high fields, thereby ensuring integrity for all high fields belonging to the `High` set.

Examples

We illustrate specification of confidentiality with JML in the first example. The example has two methods `m1` and `m2` with two fields declared. The high field belongs to security level `High` and low to security level `Low`. According to the pattern definition `m1` and `m2` maintains confidentiality but `m2` does not maintain integrity.

```
int high, low;  // high:H, low:L

/*@ normal_behavior
  @       requires true;
  @     assignable low;
  @        ensures low == \old(0);
  @*/
public int m1() {
   low = 5;
   low = low*low;
return low;
}

/*@ normal_behavior
  @       requires true;
  @          assignable high;
  @        ensures high == \old(low);
  @*/
void m2() {
   high = m1();
}
```

Figure 4.7. Example with both methods maintaining confidentiality.

Confidentiality is maintained because in both methods the flow in high fields is independent from the low fields. In the m2 method we have a flow from a low to a high field, which is not a breach of confidentiality because public data can be read by a higher security level. This however, is a violation of integrity, because the high fields are updated by low ones which have lower security level and this is a violation. Thus, method m2 does not respect the definition of the second pattern.

In the second example we have an interdependency between methods. We have method m3 and m4, where m4 internally calls m3. There are only two fields, high and low which have security level High and Low respectively.

Whether method m3 is confidential depends on the parameter i. In case i is a low field the method does not leak information, otherwise, it leaks information of a high field via its return value.

In m4 we see that there is a dependency between the low field and the value of high in the pre state. This represents a violation of the second pattern in the ensures clause. So, we conclude that the method m4 leaks information.

```
int high, low;  // high:H, low:L

/*@ normal_behavior
  @      requires true;
  @    assignable high;
  @      ensures \result == \old(high) &&
  @                         high == \old(high + 1);
  @*/
int m3(int i){
   high = high + 1;
   return i;
}

/*@ normal_behavior
  @      requires true;
  @    assignable low,high;
  @       ensures low == \old(high) &&
  @       ensures high == \old(high + 1);
  @*/
void m4(){
   low = m3(high);
}
```

Figure 4.8. Dependencies appearing when we have method calls.

4.3.1. Issues

The main limitation of this approach is that it *does not scale well* on an application with 'real life' requirements. To apply this on an application it would require a very large number of security levels making the security lattice complex. Defining and maintaining the flow between a large number of security levels would be cumbersome.

It is very hard to develop an application where the flow of data occurs within static security levels as is suggested in this approach. Another problem here is that no (controlled) release of confidential data is possible. We know that this is often a crucial functional requirement of applications. Very often, (secret) data that is declassified needs to be released. This represents a problem for this approach because such a declassification represents a violation of the main principle on which this approach defines confidentiality.

After presenting the approaches that have been taken in the past for specifying security properties with JML in Java programs, we now explain the approach that we will use to specify security properties in the application of the case study examined in the next chapter. Our approach is more similar to the second approach where the Java Card applets are presented with finite state automata's.

4.4. Our approach

The method that we will use to specify security properties in our application models the program behavior implementing those properties in a finite state automaton. The states and transitions of this automaton are expressed in JML annotations. We do implementation of our programe mirroring this automaton. All the states and transitions defined in the automaton are also defined in the programe. We show that the states and their transitions defined in the automaton also occur in the implementation of the programe.

We start by choosing the security property Φ that we want to implement in our programe `P`. We informally explain how the security property Φ should be implemented in the programe `P`. Since we will use this approach in the case study that we develop in the next chapter, we explain the architecture and the solutions employed in the implementation that should establish Φ in the given architecture. Solutions in this case represent the features and mechanisms of our programe that should establish the required properties.

Speaking in general terms, depending on the property that is chosen, different solutions are necessary to establish different properties. For example, in the event of establishing confidentiality to ensure a secure communication between two communicating parties, for example, an encryption algorithm would be implemented. In this case, the behavior of the entire programe flow, and not just the implemented protocol that ensures this property would be explained.

After explaining the behavior of the programe that should establish the property Φ in terms of the program flow, we model this behavior in a finite state automaton, denoted by `S`. The modeled behavior will actually be an abstraction of the programe `P`, capturing its overall flow that establishes that property but at the same time, abstracts away from the low-level

implementation details of the implemented mechanisms that serve to establish that property.

We implement the program flow of `P` based on the behavior modeled in `S`. To ensure that this behavior is implemented *correctly* in the programe `P`, we translate the transitions between states in `S`, into methods which implement the behavior of `P`. These methods are then specified with ordinary design by contract pre and post conditions.

The states defined in the automaton are expressed in the programe P by a ghost variable. To ensure that the method execution in P follows the same order as the transitions of states in S, *class constraints* and *invariants* expressed in JML are inserted in P. These are predicates which have to be maintained by all methods. JML statements are to be understood as predicates which should hold for the associated Java code (17).

We show how the states and their transitions defined in `S`, closely follow the execution flow of the programe `P`.

Since we previously argued that the behavior modeled in S meets the property Φ, then we can claim that the programe P also meets that property, since all the states and transitions in P are identical as in S.

Discussion

Note that during this thesis work, we show that the program flow in P is actually identical as the state transitions in S, but we do not use any static analysis tool to verify that the annotations hold for every execution path of the programe P. This is the case because this of thesis work concentrates on methods (principles) of how to *specify* high-level security properties with JML, but does not include work on verification of those properties. The focus of our work has been to find ways to express these high-level security properties with JML.

Even though, we are aware of the importance of the two going together, the verification part is left out for future work.

During our work we use the open source *plugin* named JML2[14] for Eclipse IDE that does r*untime assertion checking*. Details on downloading and configuring Eclipse to run this plugin are included in the Appendix **XX1.2**.

This tool checks JML specifications such as preconditions, normal and exceptional postconditions, invariants, and history constraints during runtime (13).

4.4.1. Comparison to other approaches

The closest to our approach, perhaps is the work done in (42). Here they specify security related properties of a Java Card applet. They design a simplified version of a client side part of the applet, and specify some of the security properties which are related to the specific architecture of Java Card applets.

They're applet is implemented in one class. Because of this, most of the JML limitations do not pose a problem for them (see Section 3.1.7 for more details on JML limitations). As a

[14] The plugin is developed and maintained at Swiss Federal Institute of Technology in Zurich, Switzerland.

result, they were able to clearly define the states of the object created from that class in one state automaton and implement the program based on that automaton.

One difference is that they're implementation maintains the state with an ordinary variable. This variable is updated as the state of the object changes during the execution of the program. Their application is designed very closely to the automaton. It contains the state and the state-transitions implemented as methods.

In our program, we use a ghost field to ensure the state of the application. No hard-coded state is implemented in the programe itself. Instead, this ghost field is used to maintain state. This field is also used to ensure the flow of method execution.

Note that in our case study, the application will require multiple classes. This gives a significant impact of the JML limitations in the specification process in our application. This impact will come with the price of the assumptions that we will be forced to make, in order to model the behavior of our application comprised of multiple classes in a state automaton.

Because of the limitation that JML annotations were designed to capture behavior of code in *one* class, when we have design decisions spreading in multiple classes, as is often the case with the implementation of security features, this makes it harder to capture and express these features with JML.

In the next chapter we proceed by analyzing the case study of a medical clinic and deriving the security properties required to secure this application.

Chapter V

5. Case Study analysis

This chapter presents a case study for a small medical clinic where we develop a prototype application in Java. This application will be used in the next chapter to specify its security properties with the approach just defined in the previous chapter.

The work on this case study starts by first doing a threat analysis for the application, where we look into what are the assets that are important to protect. After defining the assets, we turn to identifying the possible threats against those assets.

We establish the *attacker model* which looks into how these threats can be turned into valid attacks for our application. The attacker model helps us see *how* these attacks can be carried out in our application, therefore giving us a view of the protection measures required. Based on the threats and the possible attacks that we listed, we look into the security goals that we need to pursue in our application.

These goals come in the form of *high-level* security properties that our application should have in order to achieve the level of security that we want it to have.

From the overall properties that we identify, we choose a set of them to implement in the application. At the end of this chapter we explain the design decisions and implementation technologies used to achieve those properties in our application.

We use the architecture set-up of the prototype developed for this case study as an example application, to explore how well can high-level properties be specified with Java Modeling Language, using our approach from the previous Chapter.

5.1. A general overview

In this section we present the problem statement of the case study which gives us a short description of the problem. We then list the functional requirements of the application describing what our application should do and afterwards, we present the basic design decisions of our architecture and a discussion on features of the architecture layout that we have chosen.

Problem statement

A small to mid size medical clinic requires an information system for their everyday work. The number of employees using the system is estimated at 20-30 with the prospect of growth of up to 20% in the upcoming 10 years. The users of the application are classified in two categories: the medical and administration personnel.

Each category of employees has separate duties with no overlap between them. The new (digital) system is intended to replace their manual paper-based system of data administration that the clinic currently has in place. The new information system should have the functionalities of the old paper based system. This information system should have limited access to registered users only. Registered users should only be able to access functionalities which fall under their authorization.

The medical clinic operates in at most seven locations inter-connected via ordinary TCP/IP connection. Thus the system needs to be a distributed application operating from these locations and enabling its users a centralized access to data.

5.1.1. Functional requirements

The required functionalities for this information system are the following. The users need to be able to:

a. add new patients
b. find a patient
c. edit personal data of the patient
d. add new medical records to patients history, and
e. list all medical history records of a patient

There's a clear separation between what the admin staff can access and what the medical staff can access. The admin staff can work on patient's *personal* data, while the medical staff can only work on the patient's *medical* records.

Personal data include personal information belonging to the patients identity. They include the *CPR* ID of the patient, *name*, *date of birth* and *address*. The medical records on the other hand include data related to the medical history of the patient. They include: *test name*, *remarks/comments* related to that test, *date when the tests were done* and *costs* of the tests. More on this will be discussed in the implementation section 7.1.

In respect to the functionalities listed below the system should also:

a. grant access to authenticated users only
b. the registered users are assigned privileges to perform corresponding operations/tasks. Only if they have required permissions assigned to them, they are allowed to perform the requested task.

The distribution of authorization privileges is done for each user separately in accordance with company policy. The company policy separates users in two categories: *administration* and *medical* staff. As we have stated previously, the administration staff is granted privileges to perform of a), b) and c), while the medical staff is granted permission for d) and e).

5.1.2. Design choices of the architecture

This application will be built on the principles of the *client/server* architecture with client software installed in the workstations in the branches of the clinic. The clients should have minimum load and complexity while the server implements the core functionalities.

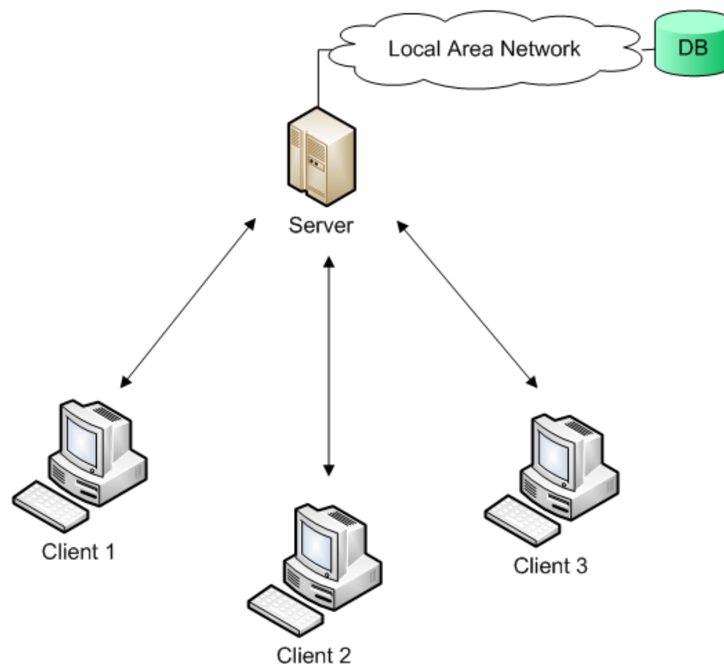

Figure 5.1: The system architecture.

The clients are connected to the server who in turn is connected to the database server that provides the centralized data storage. This connection is shown in Figure 5.1.

The database server resides on a separate machine located on a controlled network such as a Local Area Network. This machine has no public access to the internet. The clients are connected to the server through the Internet. The server is connected to the database on a dedicated LAN connection.

After initially giving the functional requirements of the application and afterwards a high-level overview of this application, we proceed by discussing the choices we have made for the architecture.

5.1.3. **Discussion**

The *client/server* architecture is chosen particularly because it is feasible to handle potential changes of the functional requirements that might come in the future. In case this is required, since all the functionality is implemented in the server, we only need to update the server in accordance with the *new* given requirements. A downside of this layout is that if we concentrate all the functionality on a single server station, this introduces the concern of a single point-of-failure in our architecture. If the server is down the whole system cannot offer services.

Another approach to this would be a distributed approach where parts of the functionalities are spread in multiple nodes offering services that are independent but complimentary to each other. This type of architecture would require multiple stations to communicate and coordinate in order for the system to service its users. In case any of these stations are out of service then only its residing functionality will be unavailable and the rest will function normally.

The database facility is placed in a local area network without access to the internet thus the database cannot be accessed by the outside world. Only the server machine can access it. This restriction will most likely be enforced by an implemented firewall and is intended to limit the penetration attempts from the outside adversaries.

5.2. **Use cases**

In this section we present the identified used cases including the actors involved in the use case diagram. Use cases are derived from functional requirements listed in section 5.1.1. The diagram is presented in Figure 5.2.

After initially giving the functional requirements and a high-level overview of the application we proceed to the next section where we analyze the threats of the system.

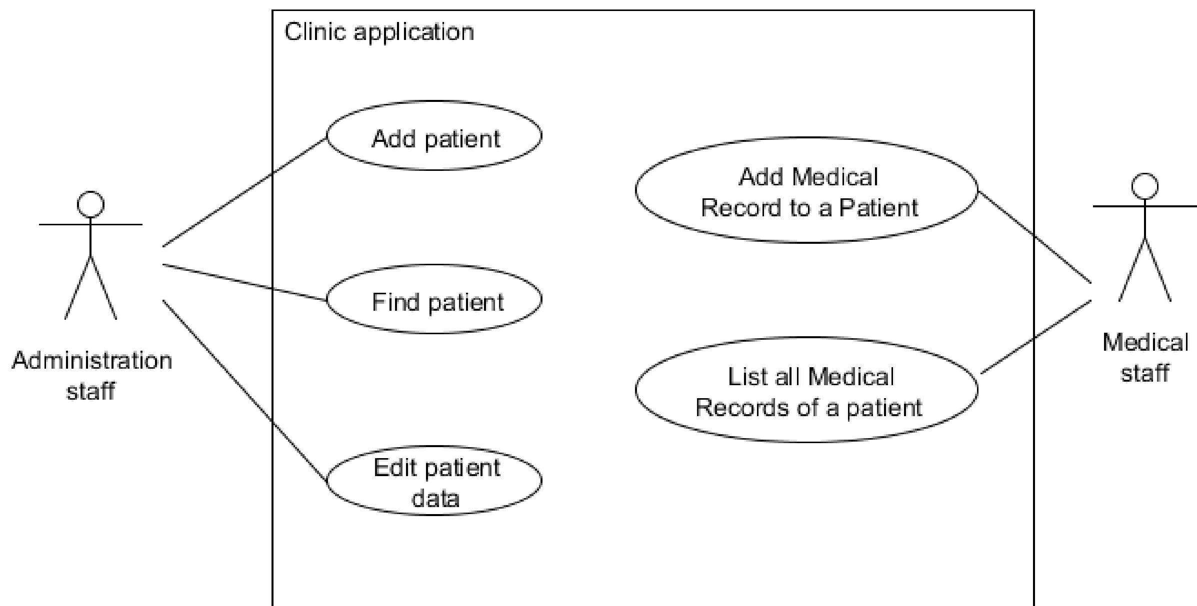

Figure 5.2. The diagram representing use cases and actors involved.

5.3. Threat analysis

In this section we identify assets which are important for the security of our application and list possible threats against the security of those assets. Threats serve as a basis for establishing security goals of our application.

5.3.1. Assets

The integrity of *data* is the main asset to be protected in information systems. A crucial concern for us is maintaining the integrity of data while it is being processed in any of the nodes of our architecture and during the exchange of data between those nodes. The workstations of our system which are considered as untrusted because of the amount of malware in circulation today. The data exchange that needs to be protected in our architecture occurs when client machines send requests and receive responses from the server, and when the server accesses the database.

In light to this, we identified two assets that require scrutiny in our threat analysis:

a. data flow

b. integrity of the data in the database server

5.3.2. **Threat model**

The threats that we have identified in regard to our assets will be presented in several categories. They are listed in categories based on what asset they target. We have grouped them into three categories based on what they target. The listed threats can target the flow of data in the application itself, the integrity of the data at the storage site or the functional requirements our application.

Category one

The first categories lists threats that are predominantly common for most of the distributed applications and in regard to asset `a` of section 5.3.1. These threats appear mostly due to the nature of distributed applications where components are spread in different remote locations connected via internet connection.

In this category we identified the following threats regarding the asset `a`:

- communication between parties occurs in an *open* network exposing our asset to the possibility of different vulnerabilities such as:
    - *sniffing* - reading the data by an unauthorized party
    - *tampering* - changing the contents by an unauthorized party
    - *message removal* - In this context message removal refers to reading the data, identifying a particular message of interest and removing that from the rest of the traffic
- identity of the parties involved in the communication process can be spoofed
- resource exhaustion can be achieved by sending a large number of fake connection requests from multiple locations to the server
- different malware programs can be running in the nodes that host our application and compromise it

Category two

Category two present threats related to asset `b`. Threats related to database protection usually come from two areas. They are connected to: *how well has the system where the database resides been hardened* and *how well is the access controlled to the database*.

The main threats coming from these areas are listed below:

- database server penetration
- excessive privilege abuse
- theft and fraud of data
- the inference problem

The main threats (47) coming from these areas can cause:

- The loss of the integrity of data through unauthorized modification. For example an unauthorized modification of the medical records of patients
- The loss of availability of the data because the database in no more operational thus the server has no access to the data and the entire system is out of operation.
- The loss of confidentiality of data(also referred to as the privacy of data), for example by an unauthorized access of patients personal data

Category three

In the third category we present threats that are more specific for our case study. These threats are directed towards compromising the functional requirements of our application. They explain threats of the application being used inappropriately.

These threats include:

- attackers can attempt to gain access to the system without authenticating
- legitimate users can attempt to perform functions which are outside of their domain

5.4. **Attacker model**

In this section we determine the possible attacks to the system based on the previously identified threats. We research how the previously named threats can be turned into actual attacks that would compromise our application. Our analysis classifies the potential attackers in three groups based on which asset they target.

The first group includes the Dolev-Yao (48) model of attackers which have full control of the network traffic. This type of attacker is able to eavesdrop on the entire network traffic, modify, replay or fabricate messages, as well as analyze the network traffic. Among the most known attacks is the *man-in-the-middle* attack where the attacker impersonates the identity of the other party involved in communication and tricks him into receiving all his traffic. Another attack is the *Denial of Service* attack when the server receives a large number connection requests from multiple locations. All its resources are used to deny these invalid connection requests and no resources are left to establish communication with valid requests. The only limitation of the Dolev-Yao attacker is that it is unable to break existing cryptographic protocols.

The attackers of the second group exploit weaknesses coming from the threats of category two in section 5.3.2 related to the integrity of the data in database server. If the current patches are not installed on the database server it is relatively easy for attackers to learn the weaknesses of the software that has not been updated to the latest patches. This offers them

an opportunity to exploit the weaknesses and perform attacks that the newest patches would prevent.

If access control privileges are not regulated properly and users have unnecessary privileges exceeding their requirements then they can abuse them and cause unauthorized modification of data or even destruction. For example if ordinary users are given administrator privileges in accessing the database then they have permission to do much more than they usually need and in this way they can seriously compromise it. For example they can manipulate the access control lists of the database.

Even though communicating between the server and the database is done on a dedicated connection the database connection strings need to be encrypted during transmission as these strings can be abused by attackers. With connection strings at hand attackers that have access to the local area network can attempt to connect to the database as regular legitimate users.

SQL injection can be attempted by attackers thus data should only be accessible through stored procedures as they provide another layer of data access control. This attack is performed when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. It is an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embeded inside another (49).

With the emerging of more sophisticated data mining tools in the recent years the *inference problem* (50) has gained increasing attention. As a result of this problem the interference of data in the database occurs. Interference is the process of users posing queries and deducing unauthorized information from the legitimate responses that they receive (51).

For the third group of attacks we assume that they have physical access to the work stations running the application. Attackers belonging to this group are clinic employees who are not listed as registered users of the clinic information system and yet, they attempt to gain access to the system. An example attack for this model could be performing a dictionary attack trying to guess the password of any of the user names that they have previously acquired for example by social engineering.

Another case of attacks of this group can be the attackers that are legitimate users of the system in the clinic but attempt to gain access to functionalities which fall outside of their domain of authorization. For ex. an administration employee attempting to edit medical records of a patient – a functionality exclusively reserved for the medical staff.

5.5. **Security goals**

Based on our threat analysis and the possible attacks that we have listed we set the security goals for our solution. Every goal represents a security property that we want to achieve in our application. Security properties tell about the security-related features of our application. In other words, security goals are the standards that we impose on the behavior of our application in order to ensure that our application is secured.

We will divide the security goals of our application in two groups. The first group concerns the goals for ensuring security of the application running in the clients and server ma-

chines and the second group of goals concerns ensuring security in the database storage. In our case the application authenticates and authorizes users, but the application itself has to be identified to the database (for ex. by a database password).

5.5.1. **Application security goals**

In order to achieve security in our system we extract several goals that need to be implemented in the architecture of our application. Goals relevant to the security of the application and its hosting platforms are listed here. The goals are extracted from the identified threats in section 5.3 and presented in the following.

a. **Confidentiality**: Data exchange between parties involved in the communication process has to be kept secret from intruders that are able to read and analyze traffic in an (open) network. Data can be read during transmission by authorized parties only.

b. This goal counters the threat defined in category one resulting from the communication occurring in an open network.

c. **Integrity**: Data integrity in general is defined as safeguarding the *accuracy* and *completeness* of information[15]. In this case if data is altered during transport in an unauthorized way, this change has to be detected.

d. This goal counters the threat data communication occurs in an open network, and as such is vulnerable to sniffing, tampering and other forms of unauthorized message alterations.

e. **Mutual site authentication**: If party A is communicating with party B, a mechanism to verify the identity of the two communicating parties has to be in place. In other words, if party A is communicating with party B, they have to mutually be sure that they are actually communicating with each other and not with a third party.

f. This goal secures the system from the threats of identity spoofing between two communicating parties.

g. **Availability**: The system should be up-and-running for users who require its services.

h. This goal protects the system from the threat of resource exhaustion identified in the group of threats in category one.

i. **Authentication**: Access to the application is permitted only if the user provides proof of a legitimate identity.

j. **Authorization**: Authenticated users are allowed to perform certain actions only if they are previously assigned corresponding rights from the security policy of the company.

[15] A definition given by International Standard Organization in the ISO-17799 standard: www.iso.org

5.5.2. **Database security goals**

To ensure security of the data in the database facility we establish several goals that we require to be implemented in the architecture of the data storage system. Notice that these goals tremendously resemble the security goals which deal with the security of the application because they both have to do with data protection. The only major difference is the context and the environment of where this data protection occurs.

a. **Access control**: Only properly authenticated requests to the server can add, edit or delete the data from the database
b. **Confidentiality**: The data in the database can only be accessed by the authorized server requests.
c. **Reliability**: Database has to be up and running providing data to the application upon its request. The copy of data that the application works with as production data when it accesses the database has to be current (the latest update).
d. **Integrity**: Refers to the requirement that information needs to be protected from improper modification. Modification of data includes creation, insertion, modification, changing the status of data, and deletion. Integrity is lost if unauthorized changes are made to the data by either intentional or accidental acts.
e. **Recovery**: The database server is protected by a firewall and should be mirrored into a separate server. The application forwards queries to the mirror who hosts production data. The data from the mirroring server is periodically copied back to the 'real' database server (behind firewall). In case of corruption, this enables us full recovery of the data as it was in the latest state before it was last saved.
f. The data in the databases server should be mirrored into separate server(s). The application server interacts with the mirror(s) to answer queries from his clients. The actual databases server is periodically synchronized with the mirror(s). It should be protected by a firewall (or, if possible stored on a different perimeter – different LAN). This enables us to be safe in case the application server and the mirrors are corrupted. We can still do full recovery of the data in the state from when it was last synchronized.
g. The database server should not offer 'live' data. The data in the database should be mirrored to separate servers that are connected to the application server. The database server where data is stored is synchronized with the *mirrors* every fixed period of time. The application works with the mirrors of the 'real' data. This enables us in case of corruption full recovery of the data in the latest state before it was last saved.

5.6. **Scope of implementation**

The system is a three-tier client/server application[16] presented in Figure 5.3. It is composed of the *presentation tier*, the *middle tier* and the *data tier*. The presentation tier presents data to the user and permits data entry though the user interface. The middle tier or commonly referred to as the application server, implements the application logic and interacts with data tier. The data tier, or data storage, represents the database management system for storing data.

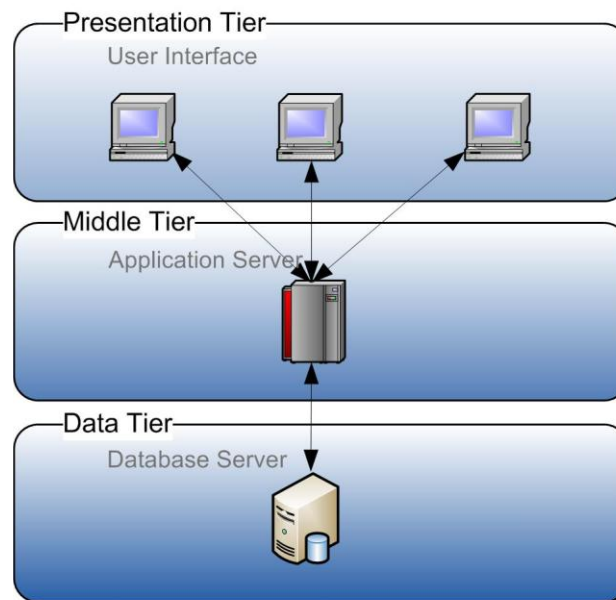

Figure 5.3. A three tier application.

Our scope of our work will include implementing the application logic in the application server. Developing the data and the presentation tier is not in the scope of this thesis work. In the middle tier we also leave out of the scope the implementation of the module that connects the application server to the database management system. This module allows the application server to query and update the database.

On Chapter 7 acceptance tests of our application are provide. For the purpose of (temporary) storing data, we use a simple data structure presented in Section 7.1.

We implement the application logic in the application server were the identified use cases are implemented.

Regarding the implementation of security properties in the application, the scope of this work includes implementing solutions for establishing the properties of *Authentication* and *Authorization* listed in section 5.5.1 under i) and j). Ultimately, since the data layer is out of

[16] One of the primary advantages of a three-tier architecture is that, as the data storage needs grow, we can change the way data is stored without affecting the clients. The middle layer usually serves to centralize the processing of business rules for the application.

scope, establishing security properties in the database server is also not included in the scope of work.

In the next section, we list the design architecture of the application and the technologies we intend to use during the implementation of the application for this case study.

5.7. **The application**

In this section we explain the design decisions during the implementation of the prototype application. We shortly discuss how the security properties included in the scope of work will be established in this application.

It was already mentioned that this application will be built on the principles of the *client/server* architecture with all the functionality implemented on the server. The client software (even though it is not included in the scope of work) would be a simple graphical user interface (GUI) application which would validate data entries by the user, send that data to the server and present results back.

Since the full implementation of the presentation tier is not in the scope of work, and we need to provide the interaction between the user and the application (which would be made possible though the presentation tier), we provide only one simple class to compensate for this. This class fills the gap for the presentation tier and enables the interaction between the user (running the client software) and the sever. This class *statically* connects to the server, calls methods on the server, forwards data there and receives the results back. Statically refers to the fact that this is done only *once* and then the program exits. Thus, in this prototype application we do not implement a 'real' interaction that one would find in an ordinary application where the user is able to repeatedly interact with the application.

To gain access to the system the user will need to be assigned a username and a password from the administration. The administration also assigns permissions for each user about the tasks they are allowed to perform. The tasks are defined based on the identified use cases defined in Section 5.2 of this chapter. Each use case is implemented in the application server within one method and that use case can be completed by the user calling the corresponding method for that use case. So the administration basically gives permissions separately for each user, about what methods he is permitted to call.

The user first submits his login credentials in the form of a username and password from the client to the server to prove his identity. If credentials are valid he is allowed access to the system and allowed to call methods on the server. Each method that the user calls on the server is subject to verification of his authorizations.

Since we don't implement the presentation tier to do this interactively, we do this statically in the class provided to compensate the interaction between the user and the application.

5.7.1. **Technologies used**

The implementation technology used to connect the clients and the server is Java *Remote Method Invocation*. It is a relatively old technology, it has been around for almost fifteen years[17], and it simplifies remote communication by not having to deal with sockets during programming (see Section 3.2, on how Java-RMI works). This technology provides us with a high-level interface of interacting with the remotely located services as if they were local.

Since we use JML compilers to parse annotations that specify the security properties, we are forced to use Java version 1.4. At the time when we started working on our application, on May-June 2010, this was the latest version of Java that current *stable* JML compilers could work with.

To achieve properties of authentication and authorization we implement the Java Authorization and Authentication (JAAS) mechanism on the server (see Section 3.3 for more details on JAAS). So, clients now connect to the RMI server where JAAS restricts user access by checking their login credentials callers and enforces authorization control to the (authenticated) users when they attempt to execute methods on the RMI server. The concept on how this is achieved is explained below.

5.7.2. **The concept behind restricting public access to the RMI server**

We have stated earlier that once a RMI service is made available in the registry anyone can access it. Any client knowing its address can bind to the registry, obtain a reference to the object, and call its methods (52). This poses a problem for us if we want to restrict its use to only a limited number of users.

Here, we briefly introduce the concept of how we intend to use JAAS to deal with restricting access to publicly available RMI methods[18]. Namely, how we intend to restrict access to the RMI server to only the authenticated users calling methods from the client programs installed in the workstations of the clinics.

Note that the user and the server in this case are operating on two different, remotely located Java Virtual Machines. In order to control access and to allow only authorized calls to the server, we implement JAAS in the server and use it to control access to (the part of) our program which implements the functionalities required by use cases.

So, the user first connects to the server who has been configured to use JAAS, authenticates and then is able to call methods on the server based on the privileges that he has. These privileges are assigned to users individually in the JAAS policy file by the administration. Initially, when the user log's in, his (previously) assigned permissions are dynamically attached to his identity. So, when an (authenticated) user calls methods on the server, JAAS verifies if his identity has necessary permissions for that call. If the identity of the calling user possesses the necessary permissions, it allows the call (see Section 3.3 for more details on JAAS). If it doesn't, an exception is raised. The basic concept of this architecture is pres-

[17] It was first added to Java version 1.1 in the year of 1997.

[18] We are aware that the idea of exposing functionalities that we want to limit access to only a number of users, is not a smart idea, but we wanted to explore how this can be achieved when using RMI.

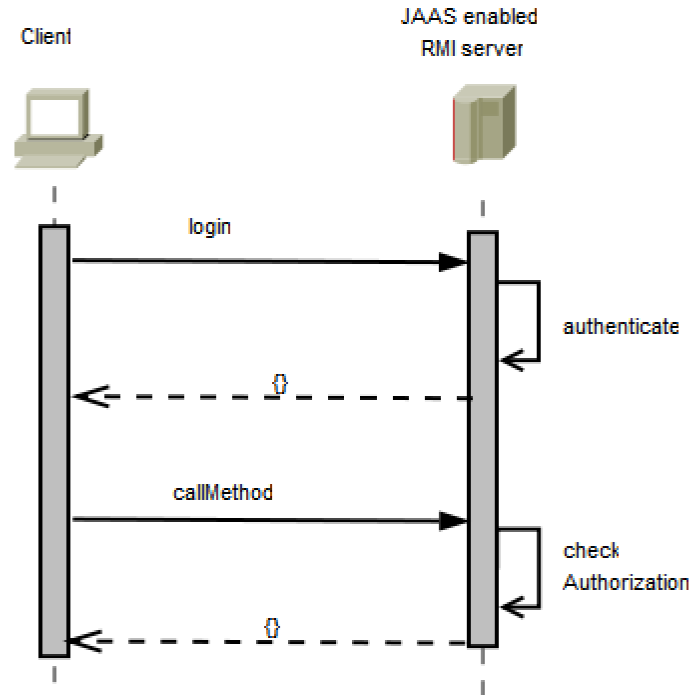

Figure 5.4. A sketch of the logic of the RMI server with access control by JAAS.

ented in the figure above. A security manager is installed in the RMI server, who verifies if the calling authority has been previously assigned permission for that call.

In the next chapter we explain some of the features of JAAS when used in this layout, we explain this concept in more detail and argue how these features will meet the properties of authentication and authorization control in our prototype application. We show the properties are specified with JML.

In the next chapter we proceed by explaining in more detail how do we intend to implement the properties included in the scope of work and apply our method defined in Section 4.4 to specify those properties with JML.

Chapter VI

6. Implementation of properties

In this chapter we explain how we intend to implement the properties of authentication and authorization in our application. We explain how these properties are achieved *functionally* in the architecture of our application. We show how our application establishes these properties by implementing the features of Java Authentication and Authorization mechanism in the architecture of our case study.

For better clarity, we present sequence diagrams of the processes that interact in the application that achieves the given properties. After explaining the intended behavior for establishing the above mentioned properties we model this behavior as a finite state automaton with defined states and transitions between those states. We do this because we want to express this automaton with JML.

This automaton is used as a model for the implementation of this case study application. The goal for modeling the behavior of the program is that the states and transitions defined are translated into *invariants* and *class constraints* expressed by JML annotations in the application. The annotations can be used to verify that the implementation of the program indeed mirrors the behavior defined in the automaton in terms of states and state transitions.

Since these states and state transitions establish the properties of authentication and authorization and; if the implementation of our program holds for the provided annotations, then we can claim that the program also establishes the properties defined in the automaton.

This approach of specifying security properties enables us to ensure that the behavior the programmer had in mind to implement security properties in the application, indeed corresponds the behavior represented in the JML annotations.

The remainder of this chapter is organized as follows. The next section gives a high level description of how the properties are established in the implementation. Section 6.2 describes this behavior with sequence diagrams. Section 6.3 models this behavior on a state automaton listing the assumptions made. Section 6.4 shows how we have specified the implementation of our program with JML annotations that that express the intended behavior and Section 6.5 discusses problems and benefits of taking this approach.

6.1. **Architecture of the application**

In this section I explain in more detail the architecture of our application and how we intend to achieve properties of Authentication and Authorization using JAAS[19]. We give some more details and background on how JAAS functions in general and how its implementation in our architecture will establish the two properties.

We have stated earlier that JAAS is implemented on the server. At the server the JAAS mechanism first verifies the identity of the caller and then checks if he has the necessary permissions to call that method. Only if the (authenticated) user has the corresponding permissions to call that method, the call is allowed and otherwise an exception is raised.

To verify the identity of the user we implement a login module which is a built-in feature of JAAS. The login module creates a *login context* for each (successfully authenticated) user individually. As part of this login context, a proxy is created also and its reference is returned to the caller. When the proxy is created it contains the identity of the user. The user uses this reference to call the methods on the server.

So, when the user calls the methods on the server, the proxy checks his permissions. If the caller has the necessary permission then the proxy calls the method on his behalf. If it doesn't, an exception is raised. The authorizations are checked at runtime to verify if the caller has execution privileges for the requested method. Each of the methods that the proxy can call completes a corresponding use case of our application.

A more detailed concept behind this approach is presented in Figure 6.1. The steps occurring when the identity of the user is authenticated and his execution privileges checked are presented in the Figure 6.1. They are ordered numerically.

Initially, the server advertises the JAAS login module object to the RMI registry system so that clients can connect to it. In step 1, users of the client software attempt to authenticate themselves to the server. In step 2, when correct credentials are presented by the user a new login context is created for the identity of that user in the server. The login context is a built-in mechanism of JAAS. In JAAS the identity of the user is represented by a `Subject` class. All permissions that have been assigned to the user in the policy file by the administration are attached to the subject representing his identity in the application. In this way after each successful user authentication, JAAS creates a login context for that user and attaches his `Permission` objects to his `Subject` class.

Next, a proxy object for that particular user (containing his identity) is created in step 3 by the login object. In step 4, the reference of this proxy is returned to that particular user, for which the proxy was created initially. So now, only he can reference the methods of the proxy because he is the only one that has the reference to it. And on the other side, the proxy knows the identity of the user as well, so it only accepts calls from the user that it was created for.

The role of the proxy is *to only check if the user has necessary permissions to call that method* while the functionality of the method itself is implemented in another object; in Figure 6.1 denoted as `Server Impl`. So, when the user actually calls the methods on the RMI server, he is actually referencing the proxy first, who verifies his authorizations and then calls that method for him.

[19] Java Authentication and Authorization Service

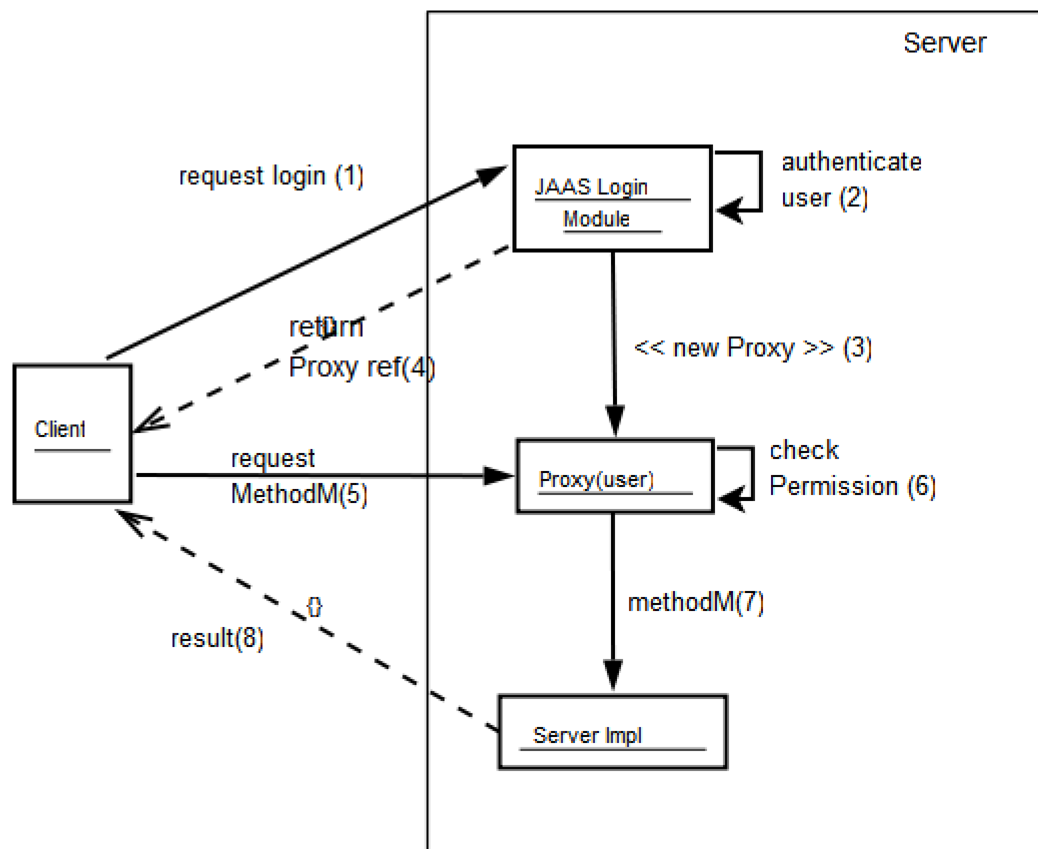

Figure 6.1. The simplified concept of restricting access on the RMI server with a proxy.

In step 5 the user requests a method by using the proxy reference which he got in step 4. When the user references a method on the proxy, the proxy first checks if that user has permission to call that method by using the built-in feature of the JAAS system. This step is denoted as step 6. Only if the user has been assigned the required permission to execute that method, the proxy calls the method on the server object where the real method functionality is implemented. This step is denoted as step number 7. After method execution, the result is returned to the client in step 8.

Note that this architecture creates a corresponding *login context* and *proxy object* for each (authenticated) user but the actual use cases are implemented is a single object listed in the figure as `ServerImpl`. So, all the created proxies verify the authorizations of their corresponding user and call methods in this object for him.

Figure 6.2 shows this scenario with two users authenticated at the same time. After their successful authentication a login context and a dedicated proxy is created for each of them. The reference to the proxy objects is returned to the corresponding clients who can after wards use it to call the methods on the server. And since only the authenticated users have references to their corresponding proxies then only they can call methods on the server by having the proxy call them on their behalf. This limits the completion of use cases on the server to *only* the users that have logged-in successfully, which is goal that we wanted to implement in our application.

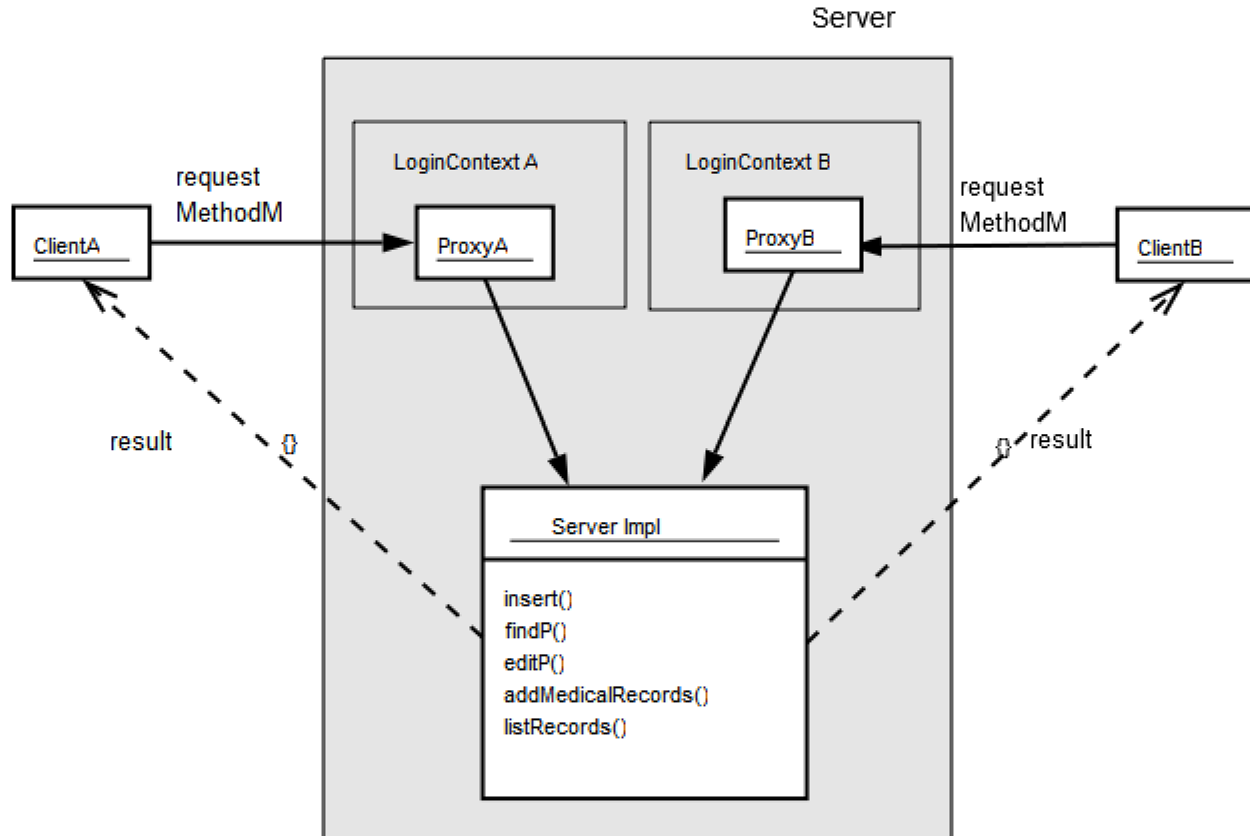

Figure 6.2. Dedicated proxies and login contexts created for each authenticated user.

We wanted to allow only authenticated users to be able to have access to the methods on the server, subject to verification of access control rights (permissions) required by the administration for that particular method call on the server.

6.1.1. JAAS features in our application

Verification of access control rights of users in our application is done by the policy system of JAAS. The policies defined are enforced by installing a security manager. Starting from version 1.4, Java implements a *code-centric* security mechanism (35). This means that permissions can be granted based on code characteristics: where the code is coming from, whether it is digitally signed and if so by whom. (for more details on how permissions are expressed in the policy file go to Section 3.3.1). By using the JAAS authorization mechanism we are able extend the existing code-centric access controls of Java with new *user-centric* access controls.

Now we can restrict execution by not just where the code is coming from and who signed it but also by identity of the user who is trying to execute that code. This is done by creating policies that list which user is allowed to execute which classes and what system resources

are these classes allowed to access. In other words, permissions can be granted based, not just on what code is running but also on *who* is running it.

How is the JAAS policy enforced?

When a Java application starts by default, it has no security manager installed. At its option, the application can install one. If it does not install a security manager, no restrictions are placed on any activities requested of the Java API; the Java API will do whatever it is asked. If the application *does* install a security manager, then that security manager will be in charge for the entire lifetime of that application. It *cannot* be replaced, extended, or changed. From that point on, the Java API will fulfill only those requests that are sanctioned by the security manager (35).

We require that application install a security manager[20]. The security manager controls the permitted operations based on security policies that we create. These policies are placed on separate files that security manager reads when it is invoked at the start of the application.

We will use three policies defined for our application. Each of them deals with a particular aspect of our application.

The first policy defined in the `codePermissions.policy` file is the most important for our application. It specifies *which* `.class` files on the server are allowed to use and create *what* resources. We split the files on server into two JAR archives, putting all the files that need to use security sensitive resources in the `server.jar` archive, and the other ones (that don't need such resources) in the `actions.jar`. The policy that we have just mentioned assigns permissions for each of the archives separately on the resources they can use. This limits the harm that the files can do to our application if they're compromised.

In the event that they become compromised by malware and tricked into doing harm to our application, their access to the system resources is restricted. For example, if the `server.jar` file is compromised by malware and tries to establish a TCP/IP connection with a remote host to copy all the data there, this is not possible, because the archive file is not permitted to make new TCP/IP connections to hosts. The security manager will not allow such actions because in our policy we have instructed him that this archive is to accept *only* connection *requests* not initiate new ones. The requests that are accepted can also only come from a fixed IP address and port number.

This policy is listed in Appendix C. Basically, here we list which sensitive resources can be created and used by the JAR archives in the server.

The second security policy in our application, enables the administration to assign permissions for users *individually*. The permissions are given in the form of *method-names* (of use cases) that each user is allowed to execute.

When permissions are assigned to the users, they map the *names of the methods* implementing use cases to the *usernames* of the staff in the clinic. (The methods that implement use cases are covered in Section 7.2.1). This policy is presented in the file named

[20] We invoke it by adding it as an argument to JVM when we first start the application. The argument required is: -Djava.security.manager

`Authorizations.policy`. It needs to be associated with the environment variable `java.security.auth.policy` so that the Login Context can create permission objects for users that it authenticates.

The third policy defines which login module to implement for authentication and is given in the file called `LoginModule.config` It specifies the name, options and the package location of the login module to be implemented. This naming of the modules is necessary because it is possible to have more than one login module implemented per application. In this case each module would impose different levels of restrictions. All the policy files are listed in Appendix C.

Our application implements only one login module. We invoke it with the option `required` so that the authentication needs to be successful in order for the overall login of the user to succeed. The login module is used to establish the property of authentication in our application.

This way of controlling which users are allowed to execute which use cases is convenient for us because it is not defined internally in the code but in a ordinary textual file that is loaded when the application starts up. Thus if changes are needed, no interventions in the code are necessary. All we need to do is edit the grant statements in this textual file.

The mechanisms and their features explained here will establish the properties of authentication and authorization included in the scope of work. They enable us to control access to the application and to verify if the (authenticated) users are allowed to execute the requested methods which in turn, implement the use cases of the application.

In the next section we explain how this *intended behavior* would look in an actual implementation. The term intended behavior refers to the functional behavior of our program with mechanisms of JAAS and its features that will establish the security properties in the implementation of our program.

We present the numbered steps of this behavior from Figure 6.1 through sequence diagrams. The sequence diagrams capture the order of processes and show how they work when these steps are actually implemented in the classes of an application.

6.2. **Sequence of operations**

In this section we explain the interactions of processes in the program behavior that should establish authentication and authorization in our case study.

Since the application is built on RMI technology first the remote object needs to be advertised in the RMI-Registry. The application starts by the client software looking up into the registry to obtain the reference of the (remote) object. The client must provide the address of the remote object he is looking for. If this remote object has been advertised previously, its reference is found and returned to the caller. This is presented in the Figure 6.3.

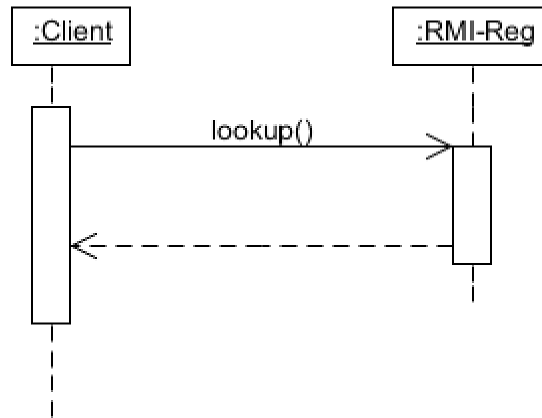

Figure 6.3. Obtaining the reference to the remote object.

The client software using this reference requests methods on the remote object of the server. The object declared as remote by the RMI system in the server implements the login module which creates a login context for the calling user. The remote object is denoted in the figure as `Login Impl` and the login context created as `Login Context`. The caller calls the built in method `login()` of the module which creates a login context and authenticates the caller. If the caller has provided correct username and password the user is authenticated successfully, a proxy is created and the reference of this proxy is returned to the caller. The proxy has now established the identity of the user.
Note that when the `login()` method is called in the `Login Context` it creates the identity of the user and it dynamically binds his permissions to his identity. So based on the grant statements that the administration has listed for that user in the policy file the object from the Subject class gets the corresponding Permission objects binded to him. This part of how JAAS functions internally is not presented in the figure. For more details on this, read Section 3.3.2. This completes the authentication phase.

The authorization phase starts by the caller requesting to execute a method on the server. Using his reference the client can now call methods on the server object which implement the use cases of our application.

Both the proxy and the other server class implement the same interface, which means that both of them implement the same methods. The difference is that the proxy object (denoted here as `Server Proxy`) only checks if the caller possesses the necessary authorizations and the implementation class (in the figure denoted as `Server Impl`) implements their actual functionality. The verification of authorization of the caller is done by executing the built-in method `doAs` of the `Subject` class. This class represents the identity of the user. This class stores all the permission objects that the user has been binded to when his identity was created. The `doAs` method looks through all these objects to find the necessary permission. If it is found then no exception is raised and the proxy calls the method on user's behalf in the `Server Impl` objects where the use case functionality is completed.

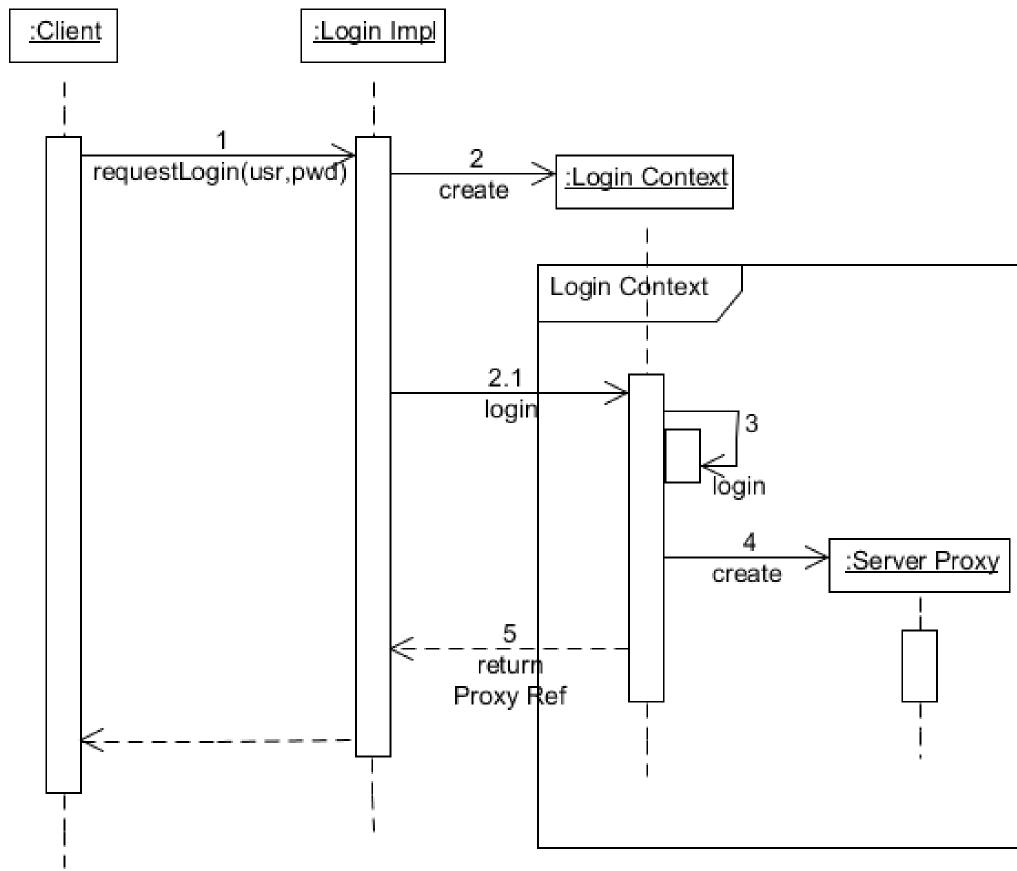

Figure 6.4. A proxy is created when the user is successfully authenticated.

An architecture that implements the sequence of steps shown in this section establishes the anticipated properties in the application.

In the next section we model the flow of these steps in a finite state automaton that captures the behavior of the application in terms of states and state-transitions. The implementation of our case study is done based on this modeling. In fact our program *mirrors* the automaton. Mirrors in this context means that states and state transitions appearing in the automata will identically appear in the program as well. To ensure that they indeed do appear identically as in the automaton, we express the states and their transitions with JML annotations.

The goal of this modeling is to show that since the application actually mirrors the automaton, and since we have showed how this behavior defined in the automaton meets properties of authentication and authorization then the implementation of our program also meets these properties.

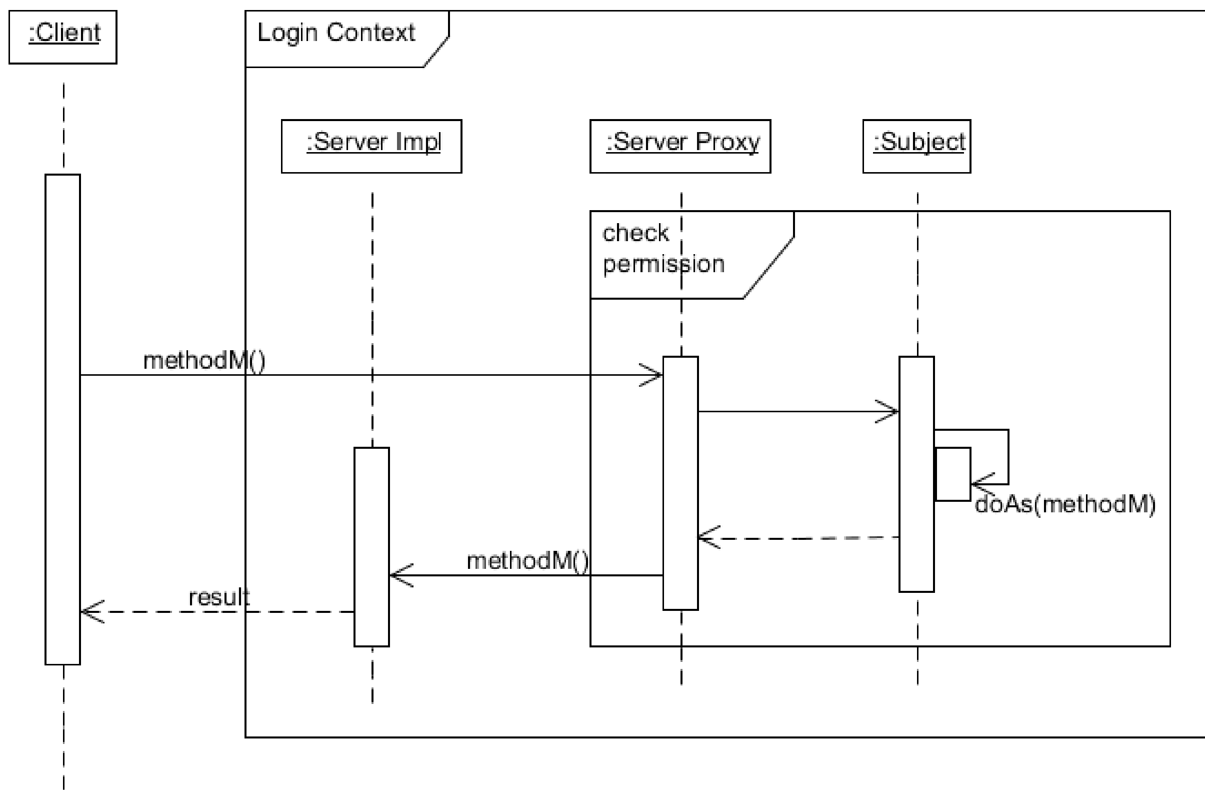

Figure 6.5. The proxy verifies the permissions of the caller.

6.3. Modeling the behavior of the application

In this section we model the flow of the program as a finite state automaton. This type of an automaton is a mathematical abstraction used to model computer programs. It is a behavior model composed of a finite number of states, transitions between those states, and actions, similarly to a flow graph in which one can inspect the way logic runs when certain conditions are met (53). The start of the finite state automaton begins in the start state and goes through the other states depending on the input that it gets from the states it has previously been in.
This automaton will capture the intended behavior of the program explained in the previous section with states and actions defined for each state. Actions here represent the *methods* allowed to be called in the corresponding states that lead to state transitions and the (possible) *exceptions* that can be raised as a result of method execution.

One of the intentions of this modeling is to abstract away from the actual implementation details which establish these properties but still be able to capture the intended (high-level) behavior of the program. Since this model is based on the abstraction of the actual application some assumptions are necessary for this model to hold. These assumptions are presented in the following section.

6.3.1. **Assumptions**

In the model of our application we include the JAAS mechanism but abstract away from how this system works internally. In the automaton we only model the requirements that a Login Context and a Subject representing the identity of the user is created by JAAS and assume that JAAS works correctly.

We assume that no unauthorized tampering of messages during communication between the client and the server takes place.

Recall that one of the limitations of JML in regard to security is that specifications are defined for each class *separately* (see Section 3.1.7 for more details). As of result of this design of JML, the static checking tools, also verify the annotations for *each* class separately. This poses a great problem for us since, the overall flow of the program that we want to model happens in more than one classes. More specifically, it begins in the `Client` class, which interacts with the `LoginImpl` and `ServerProxy` classes.

To model the continuity of state transitions between these classes into one state machine which represents the entire program behavior, when the predecessor state of the programe occurs in another class we *assume* the value of the previous state. To model this, we use the `assume` keyword of JML. By using this assumption, when the flow control of the programe goes from a state in one class into a new state in another class, we instruct the static checker to assume that the program flow came from a particular state. This assumption enables us to capture the continuity of the state transitions in the overall program.

Thus, in the case when we need to express the two consecutive transitions occurring in two different states with JML annotations, we *manually* check the predecessor state from our automaton and insert it in the programe as an assumption statement expressed in JML.

This is done so we could model the overall flow of the application in *one* state automaton. We require this because the properties that we are specifying depend on the behavior implemented in multiple classes.

The transition of states occurring in two different classes happens for example, when we go from the state when the client class calls a remote method, to the state when the server verifies the authorizations of that user (more on this will be explained when we do the actual specification in Section 6.4.

6.3.2. **The state automata**

We present the automaton modeling the life-cycle of the application in the Figure 6.6. The states are represented by ovals; transitions are represented by arrows and exceptions by broken line arrows of grey color. The automaton depicts the behavior of program occurring in the client side (marked with C) and the server side (marked with S). The dotted gray line represents the separation of the states occurring in the client side of the program shown on the left side of the automata, from the states occurring on the server side shown on the right

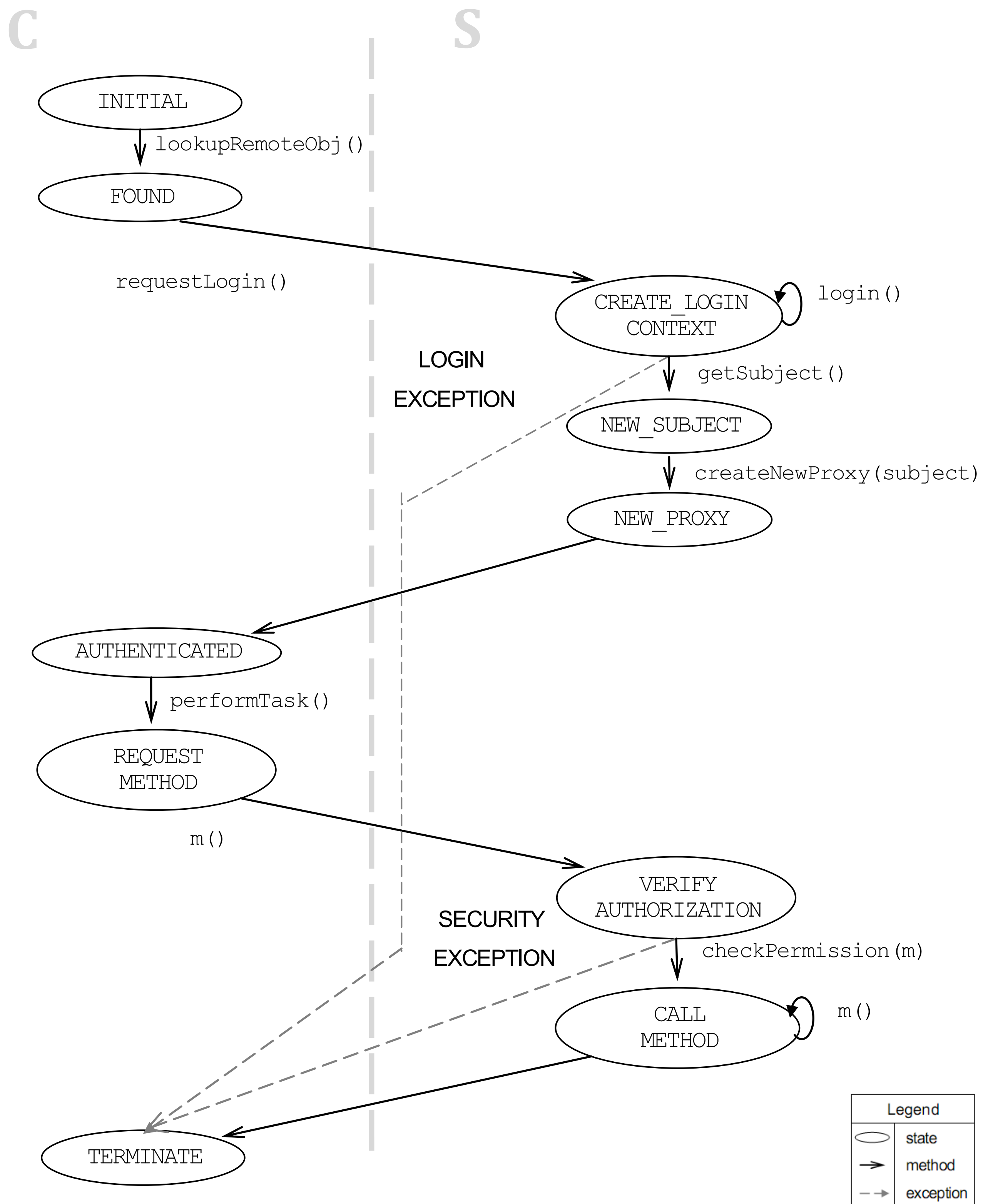

Figure 6.6. The program behavior modeled as a state automaton.

side of the figure. Since we are modeling the behavior of a prototype application, the client behavior that we model here calls *statically* one method on the server[21] and exits.
The automaton has ten states. It begins in the `INITIAL` state when the application starts. From this state the program calls the method `lookupRemoteObj()` for searching the reference of the remote object in the server which is advertised in the (global) RMI registry. If the (remote) object is found, the program goes into the state of `FOUND` and the reference of this object is returned to the caller.

From the `FOUND` state the `requestLogin()` method reads the credentials from the user and forwards them to the remote object in order to prove the identity of the calling authority to the server.

When the remote object receives credentials, it creates a new login context for the (authenticated) user; the program goes in the `CREATE_LOGIN_CONTEXT` state. JAAS verifies the credentials when the built-in method of `login()` is called. If the provided credentials are invalid then a *Login Exception* is thrown, the program gets in the `TERMINATE` state and exits.

If the credentials are valid the `getSubject()` method is executed to get the `Subject` (with its permission objects binded) for that user from the login module of JAAS. This gets the program in the state of `NEW_SUBJECT`. Then a new proxy is created for that user by the `createNewProxy()` method which puts the system in the `NEW_PROXY` state. The reference of the newly created proxy is returned back to the caller.

After receiving this reference at the client, the program goes into the `AUTHENTICATED` state. From this state, the client program calls the `performTask()` method which calls a remote method on the proxy. This brings the application in the state of `REQUEST_METHOD`. The requested (remote) method is denoted here with `m()`. `m()` in this case is just an example name for a remote method that implements the functionality of a used case in the server. For example `m()` can be a method that adds patients, lists the records of the patients or completes any other use case defined in Section 5.2.

When the proxy receives the request it enters the state of `VERIFY_AUTHORIZATION`. In this state, `checkPermission` method is called to verify if the caller is permitted to execute `m()`. If the user making the call is permitted to execute that method, then the proxy enters the state of `CALL_METHOD` and calls `m()` on the object where it is actually implemented. In this object the actual work of the used case is completed. After it is completed `m()` returns the result to the caller and enters the `TERMINATE` state and exits.

On the other hand, if the user does not have the required permissions for method `m` a *Security Exception* is raised and the program again gets in the `TERMINATE` state and exits without the user completing that use case.

This marks a complete lifecycle of the application starting from first verifying the identity of the user, checking if that user has corresponding permissions to execute the requested method and at last, executing the method.

The lifecycle defined in the automata is implemented on the program of our case study. The implementation is listed in the Appendix B. In the next section we show how we specify the defined states of the automata and the transitions between the states in the program. The states and transitions are specified with JML annotations.

[21] This modeling simplifies from a typical application behavior where the user interacts with the application by calling different methods that complete different tasks for him.

6.4. **Specification of the program**

In this section we show how we specify the properties of authentication and authorization in our program. By explicitly modeling the behavior of the programe that implements these (high-level) properties, it becomes easier to formulate the JML statements when specifying the program.

Note that we do not specify all the classes of the programe with annotations that would enable us to verify the correctness of the entire programe. Instead, we are *only* concerned in verifying the correctness of the security properties that we have implemented.
We will show that all the states and the transitions between the states defined in the automaton will also occur in the classes of our program. JML ghost fields are used to mark the states, while the order of transitions between them is translated into *constraints* and *invariants* placed in the corresponding classes where they are implemented.

Remember that when invariants are defined for the class then they must hold for any execution of the program statements. On the other hand constraints must hold at every entry point and exit point of each method (26). They may however, become broken *temporarily* inside the method but are established again when the method returns [see Section 3.1 for more].

Ghost fields are used to mark the state because, in comparison to other JML annotations which are defined before the method and can't be updated later, the value of this type of fields can be updated inside the method itself. In this way, when we want to mark a change in the state of the object (that has occurred as a result of some program flow) we update the value of this field correspondingly at that point in the code (see Section 3.1.5 for more details). The ghost field used in our implementation for this purpose is called `state`.

The complete program is listed in the Appendix B. The actual implementation of the functionalities required by the used cases is discussed in the next chapter. Here we discuss only the classes that define the behavior presented in the automata. By behavior we mean the defined states and the state transitions in our program. We show how we have modeled states and expressed transitions between these states in terms of constraints to ensure that the defined behavior is implemented correctly in the application. *Correctly* in this context means that the implementation actually behaves as specified in the JML annotations.

The complete life-cycle defined in the automaton occurs in three classes of our program. The behavior defined for the client is implemented in the `Client` class listed in the Appendix B page on page 111. This class finds the reference of the server, sends credentials of the user to there and calls methods on the server which complete the use cases for the user.

The behavior of the server occurs in the `LoginImpl` and `ServerProxy` class. The first class is listed in Appendix B on page 115 and implements the behavior of user authentication and creation of the proxy. The second class is shown in the Appendix B page 118 and implements the verification of authorizations of the user when he invokes methods.

Specification of the Client class

In the `Client` class we define the possible states for this class in the invariant below.

```
/*@ invariant
  @  ( state==INITIAL || state==CONNECTED ||
  @    state==AUTHENTICATED || state==REQUEST_METHOD ||
  @    state== CREATE_LOGIN_CONTEXT || state==TERMINATE );
  @*/
```

Figure 6.7. States of the `Client` object.

To specify the transitions between these states during the object life cycle we use the constraint listed in the Figure 6.8. With this constraint we specify the order of occurrence of transitions between the states of this class. For each state we specify what are the pre states that can lead to that state and post states that can follow from it. For example in line four we specify that prior to state `AUTHENTICATED`, the `FOUND` state must have been in place.

```
/*@ constraint
  @  (state==INITIAL ==> \old(state==null ) &&
  @  (state==FOUND ==> \old(state)==INITIAL ) &&
  @  (state==AUTHENTICATED ==> \old(state)==FOUND ) &&
  @  (state==REQUEST_METHOD==>\old(state)==AUTHENTICATED)&&
  @  (state==TERMINATE ==> \old(state)==INITIAL
  @                    || \old(state)==FOUND
  @                    || \old(state)==AUTHENTICATED
  @                    || \old(state)==CREAT_LOGIN_CONTEXT
  @                    || \old(state)==VERIFY_AUTHORIZATION
  @                    || \old(state)==REQUEST_METHOD);
  @*/
```

Figure 6.8. Constraint specifying state transitions.

Note that there are many states listed in this constraint that can lead to the `TERMINATE` state. This is because we build an RMI application where communication between the client and the server is done though the web and anything can go wrong with the connection at any of these states. These interruptions will raise a `RemoteException`, the program goes into the `TERMINATE` state and exits. These transitions are implemented and specified in our program but not visible in the Figure 6.6 for two reasons. Since they are part of the RMI functionality and not directly linked to the establishing of our properties and to simplify the representation of the automaton in the figure.

Note one important detail in Figure 6.8 which comes as a result of one of the assumptions we have made for the purpose of modeling the consecutive state transitions in the programe, when they occur in two different classes.

We have the transition of states from `CREAT_LOGIN_CONTEXT` and `TERMINATE` on one side, and the transition of `REQUEST_METHOD` to `TERMINATE` state on the other side. The two transitions occur if an exception is thrown in another class and returned at the `Client` class where it is caught. This happens in two situations: (a) when the user is not authenticated successfully or, (b) when he does not have necessary authorizations to make that method call.

Note that the state CREAT_LOGIN_CONTEXT is not defined as an admissible state in the invariant of this class in the Figure 6.7. The reason is that this state actually occurs in another class – in the LoginImpl class, when the user attempts to authenticate.

When the user makes this attempt to authenticate, the state of the program goes into the state of CREAT_LOGIN_CONTEXT but, if the authentication is not successful, an exception is returned to the Client. In this case, the application goes into the TERMINATE state and exits. To model this exception being thrown; in the code where we catch it, we instruct the static checker that the predecessor state was VERIFY_AUTHORIZATION. So this transition is covered by the assumption.

Please consult the actual code listings of these classes listed in Appendix B to see the code and the annotations provided to support the modeling of these inter-class transitions.

The application starts by calling the start() method in the constructor which sets the state of the program to INITIAL. This method implements the following behavior: it first looks for the reference of the remote object; if found than forwards the credentials that the user has provided to the server in an attempt to authenticate him, and if authentication is successful then it calls a (remote) method. If any of the two conditions is not met the application exits. The following annotation presented in the Figure 6.9 models this.

```
   /*@ requires   state==null;
     @
     @ ensures    (lookupRemoteObj() && requestLogin() ==>
     @                      \only_called(performTask) ) &&
     @
     @            (!lookupRemoteObj() || !requestLogin())
     @                           ==> \only_called(exit))
     @*/
private void start() {
       //@ set state = INITIAL;
       if (lookupRemoteObj()) {
            if (requestLogin()) {
                 performTask();
            }
       } else {
            exit();
       }
   }
```

Figure 6.9. Specification of the behavior in the start method.

The first method called by the remote lookupRemoteObj() method gets the reference to the remote object from the RMI registry.

The annotation for this method states that when this method is called the state must be INITIAL and the reference of the remote object must be *null*. There must not be a (old) reference to the remote object. By using the assignable clause we restrict that the only variable allowed to be modified in this method is the loginServer reference. It also states that during

the method execution only the `lookup` method called. By using the `\fresh` keyword we specify that that when the method terminates the result stored in the reference of `loginServer` must have been generated "freshly"; as a result of *this* particular method execution and not some other (prior) method.

In the postconditions we specify what conditions need to be met in order for the result to be `true`, and what conditions are required for it to be `false`. Note that in annotation for this method (and for the other methods as well), we don't need to catch the value of the `state` after the method returns, because this is defined in the general class constraint from Figure 6.8.

```
/*@ requires state == INITIAL && loginServer == null;
  @
  @ assignable loginServer;
  @
  @ ensures     \only_called(lookup) &&
  @             ( (loginServer==null ==> result==false) &&
  @
  @ ensures     (loginServer!=null ==> result==true &&
  @              \fresh(loginServer) );
  @*/

private boolean lookupRemoteObj() {
   . . .

   loginServer=(LoginInterface) Naming.lookup(serverObject);

   //@ set state = FOUND;

   . . .
   }
```

Figure 6.10. Specification of the lookupRemoteObj.

After the method `lookupRemoteObj()` executes, it brings the object in the `FOUND` state. In this state the `requestLogin()` method is called.

This method forwards the credentials read from the user to the server. Note that in the lack a presentation layer component for this application, we enter the user credentials statically (by hardcoding they're values inside this method).

In the specification we use the `assignable` clause again to limit that only `user`, `pass` and `theServer` variables can be modified during this method. During the method execution only the `forwardCrds` method can be referenced and the reference to the remote object should not be null and must have been updated during this method execution.

```
/*@ requires   state == FOUND && loginServer != null;
  @
  @ assignable user, pass, theServer;
  @
  @ ensures       \only_called(forwardCrds) &&
  @          ( (theServer==null ==> result==false) &&
  @
  @            (loginServer!=null ==> result==true &&
  @             \fresh(loginServer) );
  @*/
private boolean requestLogin() {
     user="admin1"; pass="admin1";
     //USERNAME AND PASSWORD OF THE user

     try {
       theServer = (ServerInterface)
                      loginServer.forwardCrds(user,pass);

       //@ set state = AUTHENTICATED;
     } catch (Exception e) {
          //@ assume state == CREATE_NEW_LOGIN_CONTEXT;
          return false;
     }
     return true;
}
```

Figure 6.11. Specification of the method that requests to authenticate the user to the server.

In this method we face the problem mentioned earlier when the state transition occurs between two different classes. This has been covered in the assumptions, where we decided that every time a state transitions occurring between two classes, we assume the value of the predecessor state.

So, in the specification of this (remote) method, if an exception is thrown in the server during the attempt to authenticate the user, at which point the state of the programe enters the `CREATE_NEW_LOGIN_CONTEXT` state, then the (remote) method forwarding the login credentials also throws an exception. After this exception, the program should go into `TERMINATE` state and exit because *incorrect* credentials were provided and the user must *not* be allowed access of the application.

To model this continuity of state transitions, going from state `CREATE_NEW_LOGIN_CONTEXT` (that occurred in the `LoginImpl` class) into the `TERMINATE` state (that occurs in the `Client` class) we assume the value of the first state in the catch clause after which the program goes into the second state.

On the other hand, if the authentication succeeds, the program enters the state of `AUTHENTICATED` allowing the user access to the application. Note that prior to entering the state of `AUTHENTICATED`, the state of the program was transformed through all the states defined for the `LoginImpl` class. Namely the programe has gone through

CREATE_NEW_LOGIN_CONTEXT, NEW_SUBJECT and NEW_PROXY states as defined in the automaton in Figure 6.6.

Now that the user has authenticated and the server has established his identity, he can begin calling methods that complete use cases.

The performTask() method requests to complete one of the use cases by calling the corresponding (remote) method in the proxy object in the server. The choice of which use case to request, is again done statically in this class.

The specification of the predecessor state occurring in another class (in this case the ServerProxy class) is also specified here. This inter-class transition occurs when the user does not have required permission to make method call that he requested; at which case an exception is thrown.

The specification of the method performTask is straight forward, not listed here.

Specification of the LoginImpl class

In the LoginImpl class the possible states are also defined in an invariant. The invariant is straight forward and is not listed here. The constraint specifying the state transitions of this class is presented below in Figure 6.12.

```
/*@ constraint
  @  (state==NEW_SUBJECT ==>
  @            \old(state)==CREATE_LOGIN_CONTEXT) &&
  @  (state==NEW_PROXY ==> \old(state)==NEW_SUBJECT) &&
  @  (state==CREATE_LOGIN_CONTEXT ==> \old(lc)!=lc);
  @*/
```

Figure 6.12. State transitions in the LoginImpl class.

In the last line of this constraint we specify that if the state of the program is in CREATE_NEW_LOGIN_CONTEXT then a new login contest must have been created.

The implementation of the (remote) method that handles credentials from the user calling the JAAS login module to verify the credentials of the user has the following annotation. Since exceptions can be thrown their behavior is also specified.

We define precisely the behavior of the method when it returns normally and when exceptions are thrown. When this method is called we require that the values of its parameters be non null, we define which variables it can modify and ensure that a login context and the Subject is created during *this* method execution (see Section 3.3 for more details about Subjects). It also specifies that when a LoginException is thrown no login context is created. The reason for this is that the user did not authenticate successfully.

```
/*@ normal_behavior
  @ requires     username != null && password != null;
  @    assignable     lc, user;
  @    ensures        \fresh(lc, subject);
  @ also
  @  exceptional_behavior
  @    requires  lc==null;
  @    signals   (LoginException le)
  @              le.getMessage()!=null&&\fresh(le);
  @ also
  @      exceptional_behavior
  @       requires    lc!=null || lc==null;
  @         signals   RemoteException;
  @
  @*/

public ServerInterface forwardCrds(String username,
  String password) throws RemoteException, LoginException {

  //@ assert state==FOUND;
     lc = new LoginContext("JAASLogin", new
                RemoteCallbackHandler(username, password));

     //@ set state=CREATE_LOGIN_CONTEXT;
     lc.login();

     Subject subject = lc.getSubject();
     //@ set state=NEW_SUBJECT;
  // Return the reference of a proxy to the calling client.

return createNewProxy(subject);
}
```

Figure 6.13. Specification of user authentication.

Note that even though the state changes twice inside this method and the sequence of this transition is not captured in the annotation, it is captured in the global class constraint from Figure 6.12.

In the second `exceptional_behavior` specification we list the `RemoteException`, to tell that it can be throw. This clause is not really useful during verification, but is given for better clarity of the method behavior.

The specification of the next method `createNewProxy` is similar and straight forward and thus not listed here. The function of this method is to create a new proxy for the (authenticated) user and return its reference to the caller.

Specification of the ServerProxy class

Since in our program a proxy is created everytime a user is authenticated, in the proxy class invariant we specify that during the life-cycle of the proxy there needs to be a `Subject`; its value must be *non null*. This basically means that when a proxy is created there needs to be a `Subject` class also in place. We also require that the reference to the object where the use case functionality is implemented is *non null*. This invariant also lists the possible states that can occur in this class. These elements constitute the invariant for this class and are presented in Figure 6.14.

```
/*@ invariant
  @     (subject!=null && theServer!=null) &&
  @     (state==VERIFY_AUTHORIZATION || state==CALL_METHOD);
  @*/
```

Figure 6.14. Defined states for the proxy.

We add a global constraint to specify the state transitions occurring in this class. Note that if the program is in the `VERIFY_AUTHORIZATION` state we assume (in accordance with our model) that prior to this state the program was in the `REQUEST_METHOD` state. Since this state occurred in the previous `Client` class in order to model the continuity of the program flow we assume that the predecessor state in that class was `VERIFY_AUTHORIZATION`. Thus, is the `assert` statement in the definition of this constraint.

So, this assumption is made to model the continuation of the program flow from the predecessor class `Client` where the request for calling the (remote) method came from.

```
/*@ constraint
  @     (state==VERIFY_AUTHORIZATION ==>
  @          assert \old(state)==REQUEST_METHOD) &&
  @     (state==CALL_METHOD ==>
  @          \old(state)==VERIFY_AUTHORIZATION);
  @*/
```

Figure 6.15. Class constraint for the `ServerProxy` class.

In this class all the (remote) methods exercise the same behavior. Namely, they first check for the permissions of the user and then call the requested method under the conditions that the user making the call has sufficient permissions. We show the specification of the `insert` method. The other methods are similarly specified. The only difference between them is the number of parameters that they take. Thus, the specification that we did for them will not be listed here.

For this method we specify that its parameters are `non null`, and only the `insert` and `checkPermission` methods can be called. The reference of the class where the method functionality takes place is denoted as `theServer`.

The method functionality is implemented in the `ServerImpl` class listed in Appendix B page 121. The `insert` method and its specification are listed in Figure 6.16.

```
/*@ normal_behavior
  @     requires  cpr!=null && fName!=null &&
  @               lName!=null && adress!=null && DOB!=null;
  @     ensures   \only_called(checkPermission, insert);
  @
  @ also
  @   exceptional_behavior
  @     signals_only   RemoteException, SecurityException;
  @*/

public boolean insert(String cpr, String fName,
            String lName, String adress, String DOB)
                 throws RemoteException, SecurityException {

   //@ set state=VERIFY_AUTHORIZATION;
   checkPermission("insert");

   //@ set state=CALL_METHOD;
return theServer.insert(cpr, fName, lName, adress, DOB);
}
```

Figure 6.16. The specification for the `insert` method in the proxy class.

Everytime a (remote) method is called from the `Client` class in the `ServerProxy` class, the proxy checks if the user has the necessary authorizations to run that method. To do this, inside of every (remote) method defined in the proxy we call the `checkPermission` method which in turn, uses the JAAS features to verify permissions of the user.
The specification for the `checkPermission` method specified the behavior similarly and defines the possible exception being thrown if the user does not have the required permission for that method. Here we specify that the `doAs` (built-in) method of the `Subject` class is called. The annotation for this method is shown below in Figure 6.17.

```
/*@ normal_behavior
  @       requires     methodName!=null;
  @     assignable     \nothing;
  @        ensures     \only_called(doAs);
  @ also
  @     exceptional_behavior
  @          signals   (SecurityException se)
  @                     se.getMessage()!=null && fresh(se);
  @*/
private void checkPermission(String methodName)
                            throws SecurityException{
```

```
   Subject.doAs(theUser,new MethodcallValidate(methodName));
}
```

Figure 6.17. Specification of the method that verifies permissions of the user.

Since our application uses RMI technology to implement the communication between the remote parties, an interface of all the remotely accessible methods has to be defined. This is a requirement of the RMI technology (see Section 3.3 for more details on RMI implementation). The interface class is then implemented by all the classes where the remote methods are implemented. In our case, this interface is defined in class `ServerInterface` listed on page 124 and implemented by the `ServerProxy` and `ServerImpl` class.

In the event that we were to specify all the classes of our application with JML, including these three classes, we could use *model fields* (54) (55). With these fields we define one (or more) model fields in the interface marked with keyword `instance`. This keyword indicates that the actual fields on the implementation class have to be linked to this variable.

All the classes implementing this interface can then relate the model fields defined in the interface with the fields that they implement (see section 3.1 for more on model fields). Other advantages of specifying interfaces with these fields can be found in (31). For a more extensive discussion of model fields regard (56).

We did not use this method, since we do not specify all the classes of our program, but only the classes that establish the security properties thus the aforementioned benefits would not help us.

6.5. **Discussion**

In this section we discuss problems encountered and our experience during specifying properties with this method. We point out the reasons behind these problems and discuss the benefits and outcome of doing property specification with this method.

6.5.1. **Problems**

The specification of properties with this method brings additional assurance that the behavior the programmer had in mind is correctly implemented in relation to the JML annotations provided. However, the fundamental problem remains; there is no formal proof of the automaton modeling this behavior.

A problem that we encountered during specification of our application was related to specifying the *remote exceptions* that the RMI methods throw when the connection is interrupted. It is not possible to express in pre and post conditions when this type of exception is thrown because it does *not* depend internally from our application but from an external fac-

tors (the interruption of the TCP/IP connection). Based on our experience, we are convinced that specifying exceptions is not useful for verification.

We have experienced that getting the specifications right is a labor intensive work. The JML2 Eclipse plugin that we used for doing run-time assertion checking does not report the possible *logical* errors in the specifications. Thus it is very easy to include annotations in the programe that are syntactically correct, but do not express any *valuable* `pre`- or `post`- condition that could be used during verification because their logic is erroneous.

Furthermore, sometimes the JML2 compiler did not report even the syntactical errors.

On occasion when we compiled the annotations with the JML2 plugin, it did not show us warnings of syntactical errors even though we were sure about them. On these cases, it helped restarting the Eclipse IDE and recompiling the annotations back again. This time, the warnings were shown.

In the past, correctness has been formally proofed on small and relatively simple Java Card applets (41) (16). Since their behavior is considerably simpler it is possible to model their security related behavior as a protocol and verify this protocol for the given properties. Despite this being possible, the properties specified do not constitute high-level properties such as authentication, authorization or alike. The properties specified there are usually low level properties which are specific to the architecture of the processing device (the chip) where these applications run. An example of a property specified here is atomicity of transaction.

It is generally acknowledged that there is a substantial gap between modeling the high level properties such as authenticity, confidentiality and similar with specifications done on the implementation level, usually consisting of `pre`- and `post`- conditions for methods and invariants or class-wise history constraints (42).

Modeling Java desktop applications with real-life requirements as protocols is however considerably more challenging because of the complexity involved in them.

When modeling the behavior of the application for our case study, a number of abstractions from the actual implementation details are done. It is generally known that, it is at the implementation level that often errors compromising security of the application appear. For example, our model of the application uses the JAAS features of login context and the Subject representation of the user identity but it abstracts away from the implementation of JAAS that creates these features. So if verification of the program is done, we are only assured that that these features are created but no assurance on their actual implementation of JAAS.

Some of the major problems when doing the specifications of (high-level) properties modeled in this fashion comes as a result of the limitations of JML [see section 3.3]. Usually design decisions related to security are (most often) implemented in multiple classes which affect *other* sections of the program. On the other hand, JML is meant for specifications of single classes thus, specifying design decisions that spread over multiple classes is problematic in JML. We face this problem in our implementation when modeling the flow of states between two classes. We go around this issue in our implementation by assuming when states change between two classes; we use the `assume` keyword of JML which basically assumes the predecessor state in the previous class.

Our experience during implementing this prototype application with states and state transitions defined in a state automaton has shown that additional complexity is added to the

application when the program statements have to be encapsulated in the methods which represent the transitions of the automaton.

6.5.2. **Observations**

The research efforts in the JML community were mostly concerned in developing features of JML that deal with sequential[22] programming in Java. As a result, JML lacks proper constructs to support the specification of Object Oriented programming in Java.

Based on the experience that we had during the specification of our programe, we assume that the costs of writing a fairly complete functional specification for program behavior is usually about the same as that of writing the code to implement it.

We think that capturing the behavior of code in the application level can be done easier if the programe is developed in a modular fashion. In this case annotations for modules can be written in separate files and static checking tools such as ESC/Java2 can interpret them.

Even though we don't use concurrency in our application, we notice that the specification of the *ordering of events* would be highly problematical in such programs. Simply because, specifying contracts between the caller and implementor using only the Hoare style logic (57) does *not* suffice to specify sequences of operations in concurrency.

6.5.3. **Benefits**

Modeling the program behavior as a state automaton enables us to easier write JML specifications for high-level security properties intended to be implemented in the application.

Writing the JML annotations contributes to a better understanding of the code in general. Because the programmer is thinking twice about the implementation details in the code, so the chances for discovering inconsistencies are greater.

Static analysis tools (such as ESC/Java2) can be used to discover programming errors occurring during implementation by using the JML annotations. This tool checks for common run-time errors for each class separately and issues warnings if violations of annotations are encountered. For example it checks if invariants, constraints, pre- and post- conditions are broken.

By specifying security properties of authentication and authorization in our case study, we give an example of how it is possible to actually narrow the gap when specifying *high-level* security properties on one side, and the *low-level* implementation details on the other side.

[22] Also referred to as procedural programming.

Chapter VII

7. Testing and other implementation details

In this chapter we explain how our application implements the functionalities defined in the case study. We show how the use cases are implemented and we do acceptance tests for them. Tables of acceptance tests are provided in Section 7.3 showing the relation between the test input data and the generated output.

Our functional testing involves identifying and testing all the functions of the system as defined within the requirements. This form of testing is an example of black-box testing since it involves no knowledge of the implementation of the system.

We test if the users are authenticated properly when they provide appropriate credentials to the system. We also test the system if it controls the authorizations of users properly when they initiate remote method calls to the server.

Our test cases serve to assure that the policy of separating the duties in the clinic is properly implemented in the application. Namely, we test to ensure that *only* users that were assigned privileges belonging to the administration staff can work with patients' personal data and, *only* users assigned privileges of the medical staff can work with patients' medical records.

In Section 7.3.3 we do test cases to see if the application implements the use cases with the requirements that we have defined them.

In this prototype application we use a data structure to store data in the memory. The details of the design choices for this data structure are explained in Section 0.

In the previous chapters when discussing the architecture of the application, we have stated that the proxy object created for the (authenticated) user calls methods for him in the `ServerImpl` class (listed in Appendix B pg. 121). In Section 7.2 we explain both the class that implements the use cases and the class of the proxy object. This section includes their *class diagrams*.

In the end of the chapter we analyse of the results that we got from testing.

7.1. **Data structure**

The data structure used in this application is comprised of three Java collections for storing data of the application. They hold the list of users that are allowed to access the system and the data related to the registered patients of the clinic. The patient data is held in two separate collections, each recording the personal data of the patient and the medical records.

The following data collections are used in the application:

- a `HashTable` is used in the `LoginModule` class to hold the *username* and *password* of the users that are allowed access to the application in the clinic. The login credentials provided by the user are searched for in this table. If they are found then that caller is a registered user and he is allowed to access the application.
- the list of patients is stored in a `HashTable`. It maps the unique *CPR* number's (of type String) with the patient objects from the `Patient` class. This collection is presented in Figure 7.1.
- the list of the medical records for patients is stored in a `HashMap`. Since the CPR number is used to uniquely identity the patient, we map this ID to an `ArrayList`. This `ArrayList` holds all the medical records for that patient and every time we insert a new medical record for the patient we just append it to this list. This collection is presented in Figure 7.2.

The patient class has fields for storing *first name*, *last name*, *address* and the *date of birth* of the patient.

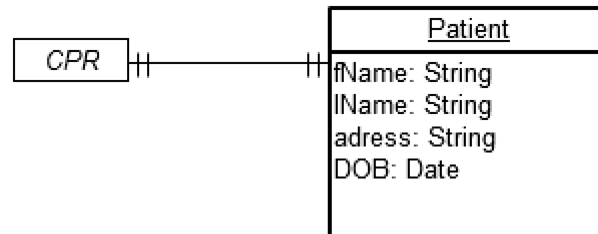

Figure 7.1. The `HashTable` for holding the list of patients.

The objects of this `ArrayList` holding medical records for patients are from the class of `MedRecords` presented in Appendix B pg. 133.

The fields of the `MedRecords` class hold the *date* when the tests were conducted, a field for additional *comments*, the *total cost* of the tests that were performed during that visit to the clinic and a list of tests done during the visit.

The list of tests for each medical record is again, stored in a new `ArrayList`. This `ArrayList` contains objects from the `Test` class. The test class is listed in Appendix B page 135 and holds the *name of the test* that was done and the *result* of that test.

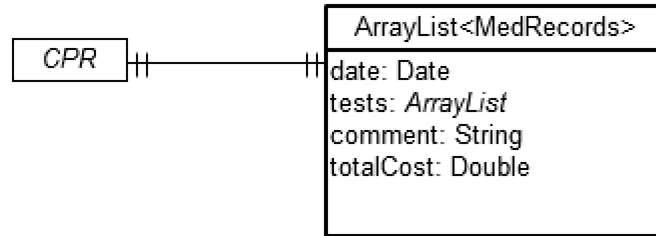

Figure 7.2. The `HashMap` for medical records of patients maps CPR to array list holding all the medical records of that patient.

The symbol ⧺ used to express the relationships between entities in the two previous figures represents a *one* to *one* relationship. s

Note that this data structure satisfies only the very basic data storage needs which are not complete for an actual real-world medical clinic. The simplification was done because the data layer for the application is not implemented in our application. Thus we chose a limited data structure with basic needs for the application.

However, in case the application is extended with an actual database, this data structure can serve as a good layout model for designing a more extensive relational data base.

In the next section we proceed by explaining some of the other design elements concerning our application. They are related to the implementation of use cases and the internal structure of class files in the application.

7.2. Design of the application

In this section we explain how the use cases are implemented. We explain the methods that implement their functionality and show the class diagram of the class where these methods are implemented. We list the internal structure of the class files of the application grouped in packages and JAR archives.

7.2.1. The implementation of use cases

Here we explain classes of the program in the server side of the application, where the required functionality for the use cases is implemented. The class diagram of the `ServerImpl` class that implements these methods is listed below in Figure 7.3.

In the previous chapters we have stated that the functionality of each use case from Section 5.1 is implemented in one atomic method. In Table 7.1 we list the names of the methods and the corresponding use cases that they complete. But first we show the class diagram of the class where these methods are implemented.

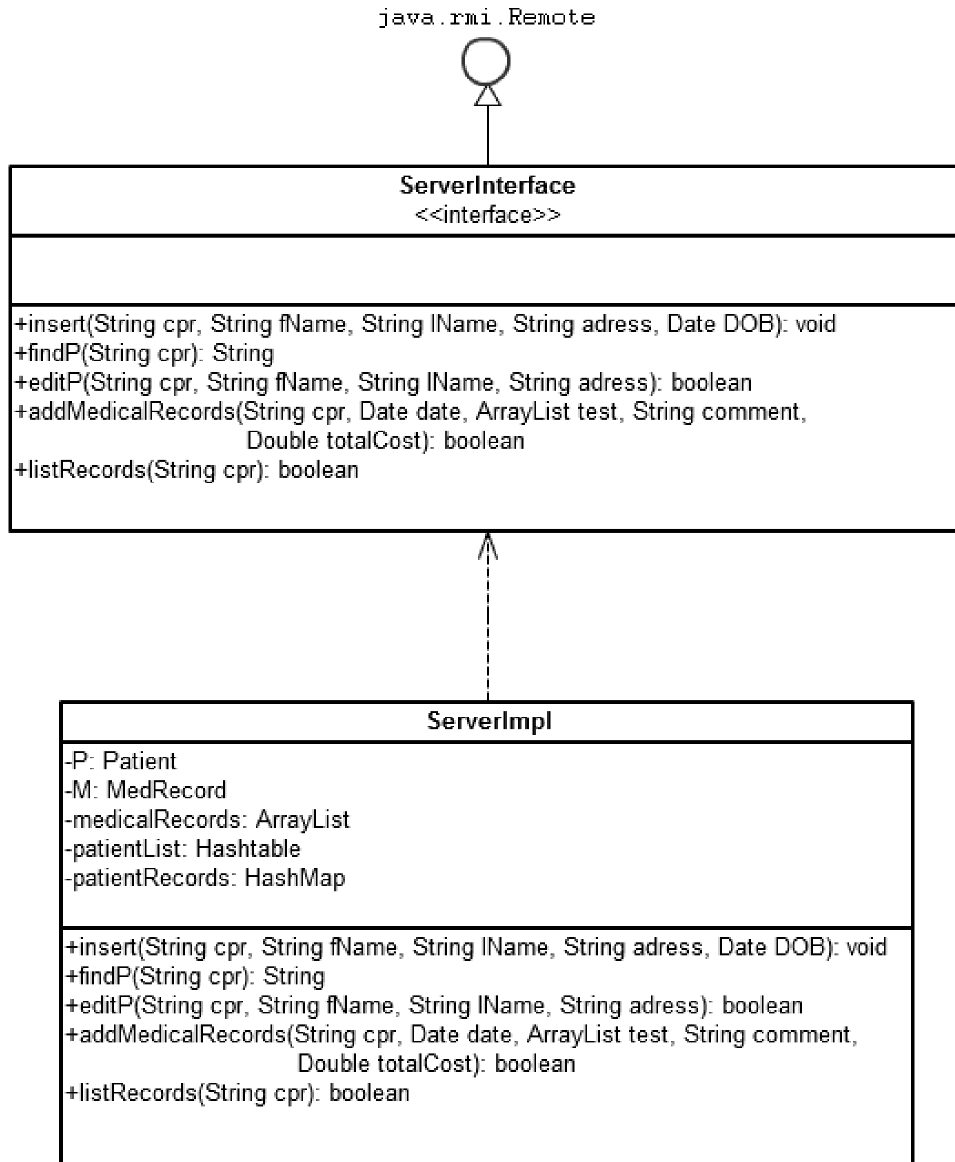

Figure 7.3. The class diagram of `ServerImpl`.

Method name	Use case
`insertP`	add new patients
`findP`	find a patient
`editP`	edit personal data of the patient
`addMedicalRecords`	add new medical data to a patient
`listRecords`	list all medical history records of a patient

Table 7.1. The method names that complete the use case functionalities.

The `insertP` method, first checks if the received CPR ID for the new patient to be inserted exists, and if it doesn't it inserts it. If the patient is inserted true is returned, and if not, false is returned to the caller.

The `findP` method returns the string representation of the patient object to the caller.

The `editP` method edits only the fields of the patient object for which the received parameters are an empty string. The caller of this method gives only values to the fields that he wishes to update. The parameters that he leaves as empty strings are neglected in the update.

The `addMedicalRecords` method first checks if that patient exists, if he exists then it checks if he has any previous records listed. If not it creates an `ArrayList` and places the (newly created first) medical record in this list. If there are records listed, it just appends to them by inserting the object in the (existing) `ArrayList`.

The `listRecords` returns the string representation of the medical records from the patient identified by the CPR id that it received as a parameter.

The Proxy class

The class creating proxies for users that are authenticated successfully is named `ServerProxy` and listed on page 118. This class implements the same interface as `ServerImpl` class. Remember that the role of the proxy is to check if the caller has the permissions for the method call he is making. This verification is done in the `checkPermission` method.

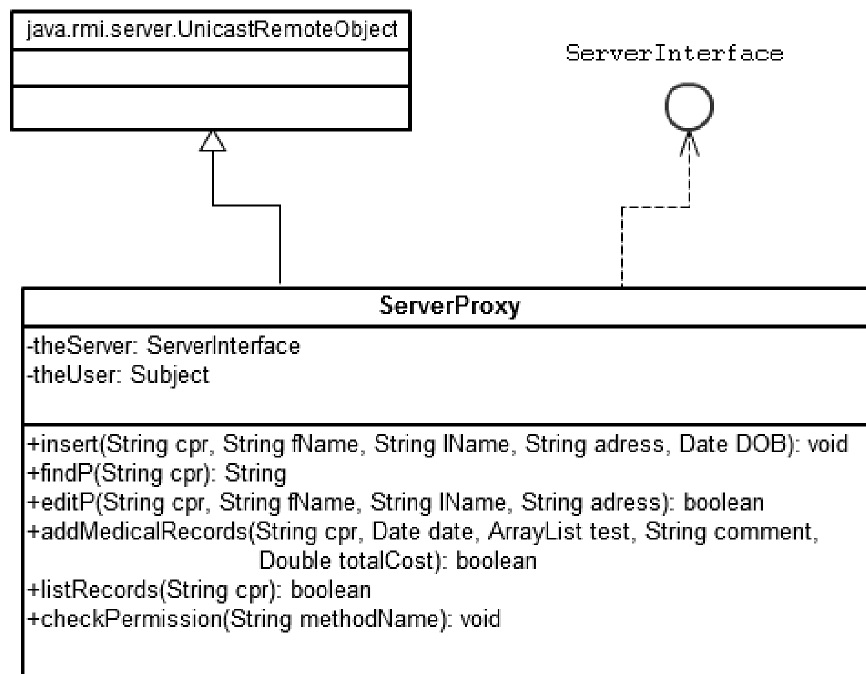

Figure 7.4. The `ServerProxy` class.

7.2.2. How is access control enforced in the server

We have discussed earlier that JAAS performs authorization control of users based on principles which in our application are defined in the `MyPrincipal` class. They are basically objects that get inserted in the Subject class which is created after the user is authenticated and signify all the permissions that the user has.

The principal objects are added to the subject class at the start-up of the application according to the policy file defined earlier and loaded then.

The principle authorization is enforced through the use of `javax.security.auth.Subject.doAs` method, which allows a piece of code to be executed with the privileges of a specific principal. The `Subject.doAs` method is called in the proxy class which requires the code to be executed as an object. For this purpose the `MethodcallValidate` class serves, where the `java.security.AccessController` calls it's `checkPermission` method to see if the Subject class (representing the user making that is making the call) has the required permission object. If the Subject has the required permission object, the code is executed and if not, a security exception is raised and the actual privileged code is never reached.

After a security manager is installed in the application, all the method calls are forwarded to the `java.security.AccessController` class who makes the final decision. This class decides, based on the principals that the user making the call has, if the code is allowed to execute or not.

7.3. Acceptance tests

In this section we present tests done on the system. We have chosen to test our application with acceptance tests. These tests ensure that the used cases of the system are implemented as required in the functional specification of the system. Using this method of testing, the application is tested from the user's point of view. We run a suite of tests providing the user-input and we verify if the outcome is as expected.

We use acceptance testing with *column fixture*. This form of acceptance testing represents input data entered in the system and the output data generated during testing in the form of rows of tables (58). Each test case is presented in a separate table. It is best used for testing functional behavior, i.e. the relation between inputs and outputs.

It is possible to use frameworks for integrated testings, such as Fitnesse[23] that does tests automatically by using software tools available online. In this case the coding of fixtures is necessary to connect the tables with the tool that executes the tests on the program being tested and reports back the results. Fixtures are the coding done by the programmer to express the HTML tables in the format that the online tool can process.

However, we did not use this framework. Instead we choose to do the tests manually, by entering the test data in the application and using print statements to read the generated

[23] www.fitnesse.org

outcome. After placing the outcome in the corresponding columns we examined it if it was correct.

The first test that we have done was to test if the authentication and authorization is functioning properly.

7.3.1. Authentication

After registering the credentials of two users in the `LoginModule` class where the authentication takes place we test if the clients providing correct credentials are allowed to login.

user	password	requestLogin
admin1	admin1pass	`true`
med1	med1pass	`true`
admin1	A	`false`
med1	MED	`false`
admin	admin1pass	`false`
Med	med1pass	`false`

Table 7.2. Acceptance test for user authentication.

From this table we see that when users enter valid credentials to the system, they are authenticated successfully (the first two cases). In the four last cases where the incorrect username or password is given, the authentication of the subject fails.

7.3.2. Authorization control for method execution

In this test case we test if the application correctly implements the of separation of duties between the administration staff and the medical staff in the clinic. The company policy states that only the administration staff of the clinic can work with the personal data of the patients, while the medical staff can operate with the data related to the medical history.

As a result of this policy the administration staff is only allowed to call methods `insert`, `findP` and `editP`, while the medical staff is allowed to call methods of `addMedicalRecords` and `listRecords`.

In our system, we register two users (one admin and one medical user) in the system and assign them corresponding permissions in the policy file. Namely, the user `admin1` is assigned permission to execute methods `insert`, `findP` and `editP` while the `med1` user is assigned permission to execute methods `addMedicalRecords` and `listRecords`.

To verify that the application allows only method calls from authorized users we do the following test. First we login with the admin user and call all the methods and then we do the same with the med user.

user	performTask	throws Security Exception
admin1	insert	no
admin1	findP	no
admin1	editP	no
admin1	addMedicalRecords	yes
admin1	listRecords	yes
med1	insert	yes
med1	findP	yes
med1	editP	yes
med1	addMedicalRecords	no
med1	listRecords	no

Table 7.3. Test results for the authorization control during method calls.

We see that in accordance with the policy, exceptions are thrown only when users call methods that they do not have permission to call.

Note that since this is a prototype application the implementation of the client software statically calls only one method and exists. So, to do this test we manually changed the reference to the (remote) method call for each test.

After testing the authentication and authorization control in the application, we proceed by testing if the application meets the functional requirements defined.

7.3.3. Implementation of use cases

Next, we test if the use cases defined are correctly implemented in the application. Note that during these test the user went through the same authentication and authorization procedure as in the previous tests, but is not shown here for simplicity. Instead, here we focus only on how methods function.
We start with testing the insertion of new patients when the `insert` method is called by the user.

Insert a new patient

In this test case we check the implementation of the `insert`[24] method. We test if we can insert more patients with the same CPR ID.

CPR ID	first name	lastname	adress	DOB	insert
0101903989	John	Doe	Street 1, Copenhagen	01-01-90	true
0101903989	John	Doe	Street 1, Copenhagen	01-01-90	false

[24] This method first checks if the patient exists and only if he doesn't it inserts it. Patients are uniquely identified by their CPR ID

0101903989	John	Hanssen	Street 1, Copenhagen	01-01-90	`false`

Table 7.4. Acceptance tests for inserting *new* patients.

From the test results we see that when we try to insert patients with the same CPR ID, those attempts fail.

Find patient

In this test case we test if the application, when provided a CPR number of the patient, can find the personal records of that patient[25].

CPR ID	findP
0101903989	John Doe Street 1, Copenhagen 01-01-90
0503979895	
0101903988	

Table 7.5. Acceptance tests for finding an existing patient.

Only when we provide a CPR ID of an existing patient, the method returns the string representation of the patient. Otherwise it returns an empty string.

Edit patient

In this test case we test the application to edit any of the personal data of the patient such as update his *first name*, *last name* or *address*. This method takes three parameters, and depending on which parameter is given a value (not equal to an empty string), that field of

CPR ID	first name	lastname	adress	editP
0093930900	Leila	Knudsen		`false`
0101903989		Stenssen		`true`
0101903989	Johan			`true`
0101903989			Elektrovej 10, Lyngby	`true`

Table 7.6. Acceptance tests for editing data of an existing patient.

the object should be replaced. The parameters whose value is an empty string are just skipped during the update.

[25] An existing patient refers to the patient that has previously been inserted in our data structure.

In the first case when we try to edit a patient that doesn't exist, false is returned, but in the last three tests when attempting to update a field of a patient that exists, they all return true.

We use the `findP` method to see if the patient object has been updated correctly.

CPR ID	findP
0101903989	Johan Stenssen Elektrovej 10, Lyngby 01-01-90

Table 7.7. The result after the update.

We see that the fields of the object have correctly changed with the values that were provided during the previous update of the patients personal data.

Adding a medical record to the patient's medical history

In this test case we test the application when the patient visits the clinic and a medical record needs to be added to his history to record this visit. The medical record that is added during each visit contains the date when the patient visited the clinic, a list of tests that were done during that visit - with each test containing the name and the result of the test, any additional comment for that visit and, the total cost of that treatment in the clinic.

CPR ID	date	tests	comment	totalCost	addMedicalRecords
0101903000	01-06-10	T1name: testA, result: 23.34	visit scheduled for next Friday	2500.99	`false`
0101903989	01-06-10	T1name: testA, result: 23.34, T2name:testB, result: 99.67	visit scheduled for next Friday	900.00	`true`
0101903989	02-06-10	T3name: testC, result: 77.0	patient is released on home care	1100.00	`true`

Table 7.8. Acceptance tests for adding medical records to the patients.

Our second test, tests the application for an insertion of a record for a client that is coming to the clinic for his first time visit (and does not have a previous history in the system). In the following we test the application when the patient comes back and has already a medical history created in the system from his previous visits.

When the client software initiates this activity, the tests are sent as (Serializable) objects placed on an array list. In the table above we denote them in short as T1, T2 and T3. They are created on the client side, placed on the array list and sent over the network as a parameter to the receiving method on the server.

We see that we get `true` when we try inserting records for patients that exist and `false` when we attempt to enter records for patients that don't exist. The first successful insertion is during the first visit, and the second one is on the returning visit to the clinic of the same patient.

List the records of a patient

In this test case we test the listing of the medical history for the patients. First, we attempt to list the medical history of a patient that doesn't exist in our system and then we list the history of an existing patient.

CPR ID	listRecords	medicalRecords.toString()
0101903000		
0101903989	medicalRecords	01-06-10 testA 23.34 testB 99.67 visit scheduled for next Friday 900.00 02-06-10 testC 77.0 1100.00 patient is released on home care 1100.00

Table 7.9. Acceptance test for listing of the patient records.

We see that after listing records of the patient that was added in the previous testing suite, both of the objects from two visits to the clinic were inserted. The first object having two tests, and the second object having one test. To see their content we show their string representation on the objects that were received in the arraylist.

As expected, when trying to list the history of a patient that is *not* registered in the system, we get the *null* value returned.

7.4. Analysis of the test results

Based on the test cases we have done, we conclude that the application correctly implements the use cases as defined in Section 5.2 (for the given test data). The application enforces access control permitting access only to the registered users.
It also restricts which methods the (authenticated) users can call, based on the execution privileges they are assigned by the administration.

Discussion on the complexity of the prototype implementation

During the development of this prototype application we developed only one class for the client side of the software. Furthermore, this class is very simple because all the actions and methods that the user initiates are coded *statically*. For example, if we want the user to authenticate with different credentials, or execute a different use case we have to manually

alter the code to do this. This class connects to the server, sends the credentials of the user, executes one use case for him and then exits.

We are aware that perhaps this is an oversimplification. However, our intention during the development of this prototype was to include only those programming constructs that directly implement the anticipated security features. We tried to avoid, as much as possible, implementing other commonly found constructs that do not deliberately contribute in the establishment of the given security properties.

This simplification was applied because we wanted to avoid adding code complexity that does not deliberately contribute to the establishing of the security properties that we model in the state automaton. With a more simple code, the automaton that models the program behavior is also *less* complex. And if the model is less complex, it is easier to see the *drawbacks* and *advantages* of the applied method for property specification, which was our goal in the first place.

Chapter VIII

8. Conclusion

During this work we did a comprehensive threat analysis from the system designers point-of-view on the case study. During the threat analysis we derive all the identified security properties necessary to secure the system. We built a prototype application implementing Authentication and Authorization properties. By abstracting the program behavior that implements these properties we were able to model the relevant behavior in a finite state automaton. States and state transitions of the automaton are expressed with JML annotations. We use a run-time *assertion checking* tool to verify the program for violations against the behavior captured in the JML annotations.

The design purpose of JML is to record the intentions of the programmer with annotations that can be used to verify the programe with the help of tools. The intentions are expressed in classical Hoare logic “contracts“. Contracts contain pre- and post- conditions expressing properties that the methods should satisfy when they’re invoked and afterward returning. Constraints are added to keep track of states of objects.

Based on our experience, we have observed that the feasibility of using JML to capture *high-level* security properties in Object Oriented Java applications turned out *worse* than expected. This is due to the design limitations of JML. Namely, JML was designed to primarily be used in *sequential* programming and provide annotations for each class *separately*. There is no support to read, use, or interact in any way, with the specifications that are defined in another class.

This became an obstacle for us, when we wanted to specify with JML annotations the abstraction of the programe implemented in multiple classes.

In our opinion, using JML for verification purposes increases dependability during software production. It contributes to a better understanding of the program behavior and a precise definition of its requirements.

8.1. **Discussion**

In the method that we have used to specify high-level security properties the assumption of the predecessor state from another class remains the greatest challenge.

When two consecutive state transitions occur in different classes, it is not possible to express the occurrence of the predecessor state as a precondition for the current state – which would allow us to model the continuity of the program flow in one state automaton.

In our work we managed to model the continuity of state transitions in one automaton by *assuming* the value of the predecessor state. During these transitions we instruct the run time annotation checker to assume the value of the state that the program was previously in.

We observe that this approach introduces scaling issues. If the model of the program that is being specified with this method has a relatively large amount of states, keeping track of the inter-class states and providing assumptions them can become overwhelming.

During the modeling of the behavior of our prototype we abstract away from the actual implementation details that establish the (high-level) security properties. However, care must be taken during this abstraction, in order not to miss, modeling the important details of the implementation which ensure the properties in the program level.

8.2. **Future Work**

We have stated previously that this prototype has only one class to substitute for the client software. This class simulates the client that is using the software by statically invoking (remote) methods. We plan on restructuring the client class by including additional methods which can be invoked from the user interface.

We plan on continuing the specification of the remaining classes of the prototype application so we could verify the correctness of all the classes in the prototype.

During the threat analysis of the case study scenario, besides authentication and authorization we have identified other security properties necessary to secure this application. From these properties, we intend to implement confidentiality and integrity in the prototype and try the approach we have used here to specify these properties. We intend to implement a protocol for these properties and try to specify this protocol with our approach.

Explore the possibility of doing *refinements* to the states of the model with the aim of increasing precision during abstraction by adding more states and implementation details in the model.

Finally, we plan to investigate the feasibility of using Petri nets (59) to represent the abstraction of our programe. Petri nets are a mathematical modeling language. They are good for visualizing processes. They show states, actions and conditions for actions to be executed. These constructs seem quite close to the ones available in the Java Modeling Language. They can also show interactions between processes.

Appendix A

D.Running the prototype

The code of the prototype application was developed on the Microsoft Windows platform using the Eclipse IDE[26]. It was compiled with Java version 1.4.2_09.

To run this prototype application, please create a *new* project in the Eclipse IDE. Add new packages in the project with same directory tree as the one we have provided. Include *all* the java files in the corresponding packages.

Put all the JAR, policy and configuration files, in the *root* directory of the newly created project.

Before you run the application, ensure that your current configuration of the Virtual Machine allows you to install a security manager.

Instruct your Virtual Machine to use the following arguments:

```
-Djava.security.manager
-Djava.security.auth.login.config=LoginModule.config
-Djava.security.policy=codePermissions.policy
-Djava.security.auth.policy=Authorization.policy
```

These arguments instruct the VM to install a *security manager* and to read the provided policies.

The JAR archives

Since this application installs a security manager, the class files are put into JAR archives[27] in order for the security manager to be able to control the system resources used by the archives. The JAR archives have to be imported in the *root* directory of the project.

[26] Integrated Development Environment

[27] Note that security policies can *only* be defined for JAR archives, and not for `.class` files.

Since we use the JDK version 1.4, we had to generate manually the *stubs* and the *skeleton* classes of the remote objects[28] in the server by using the `rmic` tool. These files are placed in the `server.jar` file.

Using the assertion checker to verifying the JML annotations

We used an Eclipse plugin named JML2 to check assertions at runtime. This plugin is developed and maintained at Swiss Federal Institute of Technology in Zurich, Switzerland.

It is freely available for download for from:

http://pm.inf.ethz.ch/research/univer-ses/tools/eclipse/

Note that we could *not* install this plugin on the *latest* version of Eclipse. We had to use the version *3.4.1* of the Eclipse IDE to use this plugin. The latest version of Eclipse was not able to detect the plugin from this link.

[28] The stub and the skeleton files unable the remote objects programmed with the RMI technology to communicate remotely.

Appendix B

E. Source code listings

Client.java

```
package src.client;

import java.rmi.Naming;
import java.util.Date;
import src.common.*;

public class Client {

   /* The Server and the port used is localhost, port:6000 */
   private static final String server = "127.0.0.1:6000";
   // Change this string if you wish to use a different port

   private String /* spec_public */ user, pass;
   // Username and pasword of the identity to be authenticated

   private static final String serverObject = "rmi://" + server + "/"
                                                   + "RemoteLoginServer";
   // Full adress and name of the remote object.

   private /* spec_public */ LoginInterface loginServer;
   // Interface to the login object that authenticates users

   private /* spec_public */ ServerInterface theServer;

   /**
    * The ghost variable used to mark the state that
    * the program is currently in
    */
   //@ private static ghost String state=null;
```

```
    /*@ invariant
 @          ( state==INITIAL || state==CONNECTED ||
 @      state==AUTHENTICATED || state==REQUEST_METHOD ||
 @             state == CREATE_NEW_LOGIN_CONTEXT || state==TERMINATE );
 @*/

/*@ constraint
 @    ( state==INITIAL ==> \old(state==null ) &&
 @          ( state==FOUND ==> \old(state)==INITIAL ) &&
 @          ( state==AUTHENTICATED ==> \old(state)==FOUND ) &&
 @          ( state==REQUEST_METHOD ==> \old(state)==AUTHENTICATED ) &&
 @          ( state==TERMINATE ==> \old(state)==INITIAL
 @                                 || \old(state)==FOUND
 @                                 || \old(state)==AUTHENTICATED
 @                                 || \old(state)==CREATE_NEW_LOGIN_CONTEXT
 @                                 || \old(state)==REQUEST_METHOD ));
 @*/


//    CONSTRUCTOR
/*@ requires state==null;
 @
 @ ensures        state==INITIAL &&
 @                      loginServer==null && theServer==null &&
 @                      \only_called(start);
 @*/
public Client() {
    //@ set state = INITIAL;
    loginServer = null;
    theServer = null;
    start();

}

/**
 * Looks up the reference to the remote object; if found attempts to
 * athenticate the user and then calls methods for him. If any of the
 * methods called by this method returns false it exits the programe.
 */
/*@ requires state==INITIAL;
 @
 @ ensures        (lookupRemoteObj() && requestLogin() ==>
 @                                          \only_called(performTask) ) &&
 @
 @                      (!lookupRemoteObj() || !requestLogin())
 @                                          ==> \only_called(exit))
 @
 @*/
private void start() {
    //@ set state = INITIAL;
    if (lookupRemoteObj()) {
          if (requestLogin()) {
                performTask();
          }

    } else {
          exit();
```

```
	}
}

/**
 * Looks up the reference to the remote object on the server
 */
/*@ requires state == INITIAL && serverObject != null
  @                                             && loginServer == null;
  @
  @ assignable     loginServer;
  @
  @ ensures         \only_called(lookup) &&
  @                         ( (loginServer==null ==> result==false) &&
  @
  @                           (loginServer!=null ==> result==true) &&
  @                            \fresh(loginServer) );
  @*/
private boolean lookupRemoteObj() {
	try {
		loginServer = (LoginInterface) Naming.lookup(serverObject);

		//@ set state = FOUND;

	} catch (Exception e) {
		return false;
	}
return true;
}

/**
 * Attempt to login the user.
 */
/*@ requires state == FOUND && loginServer != null;
  @
  @ assignable     user, pass, theServer;
  @
  @ ensures         \only_called(forwardCrds) &&
  @                         ( (theServer==null ==> result==false) &&
  @
  @                           (theServer!=null ==> result==true) &&
  @                            \fresh(theServer) );
  @*/
private boolean requestLogin() {
	user="admin1"; pass="admin1"; // USERNAME AND PASSWORD
	                                // YOU WANT TO AUTHENTICATE WITH
	// update these fields with credentials of the identity
	// you want to authenticate
	try {

	  theServer=(ServerInterface) loginServer.forwardCrds(user,pass);
	  //@ set state = AUTHENTICATED;
	} catch (Exception e) {
		//@ assume state == CREATE_NEW_LOGIN_CONTEXT;
		return false;
	}
	return true;
}
```

```
    /**
     * Call a remote method for the (authenticated) user
     */
    /*@ requires state==AUTHENTICATED && theServer!=null;
      @
      @ ensures         \only_called(insert, exit);
      @*/
    private void performTask() {
          try {
                //@ set state = REQUEST_METHOD;
                theServer.insert("cpr", "Stephen", "Nielsen",
                            "Elektrovej 330, Lyngby 2800", new Date(901205));
          } catch (Exception e) {
                //@ assume state==VERIFY_AUTHORIZATION;
                // User is not authorised for this method.
                exit();  // Exit the program
          }
          exit();      // Method completed successfully. Exit program.
    }

    /*@ reqiures true;
      @ ensures        state==TERMINATE;
      @*/
    private /* spec_public */ void exit() {
          //@ set state = TERMINATE;
          System.exit(0);
    }
}
```

LoginImpl.java

```
package src.server;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;

import src.common.LoginInterface;
import src.common.ServerInterface;

public class LoginImpl extends UnicastRemoteObject implements LoginInterface {

   // The referece to the object in the server where functionality
   // of the methods is implemented.
   private /* spec_public */ ServerInterface myServer;

   //@ private static ghost String state;

   private /* spec_public */ LoginContext lc;

   /*@ invariant
     @          state==CREATE_LOGIN_CONTEXT ||
     @                              state==NEW_SUBJECT || state==NEW_PROXY;

     @*/


   /*@ constraint
     @    (state==NEW_SUBJECT ==> \old(state)==CREATE_LOGIN_CONTEXT) &&
     @          (state==NEW_PROXY ==> \old(state)==NEW_SUBJECT) &&
     @    (state==CREATE_LOGIN_CONTEXT ==> \old(lc)!=lc);
     @*/

   // Constructor
   protected LoginImpl(ServerInterface theServer) throws RemoteException {
             myServer = theServer;
   }

   /**
    *  Allows the client to login by creating a proxy for him and
    *  returning him the reference so that he can call methods on it
    */
   /*@   normal_behavior
     @   requires     username != null && password != null;
     @          assignable   lc, user;
     @          ensures           \fresh(lc, subject);
     @ also
     @     exceptional_behavior
     @          requires     lc==null;
     @          signals      (LoginException le)
     @                                 le.getMessage()!=null && \fresh(le);
     @ also
```

```
      @      exceptional_behavior
      @       requires  lc!=null || lc==null;
      @          signals     RemoteException;
      @
      @*/
   public ServerInterface forwardCrds(String username, String password)
                                throws RemoteException, LoginException {

         //@ assert state==FOUND;
         lc = new LoginContext("JAASLogin",
                          new RemoteCallbackHandler(username, password));

         //@ set state==CREATE_LOGIN_CONTEXT;
         lc.login();

         Subject subject = lc.getSubject();
         //@ set state==NEW_SUBJECT;

   // Return the reference of a proxy to the calling client.
   return createNewProxy(subject);
   }

   /*@   normal_behavior
      @         requires    subject!=null && myServer!=null;
      @   ensures           \result==(ServerProxy(subject, myServer));
      @ also
      @     exceptional_behavior
      @         requires    lc!=null;
      @         signals     RemoteException;
      @*/
   private ServerProxy createNewProxy(Subject subject)
                                          throws RemoteException {
         //@ set state==NEW_PROXY;
         return new ServerProxy(subject, myServer);
   }

}




   /**
    * The inteface between the user input and the security mechanisms.
    * It decouples the service provider from the specific input device
    * being used to enter credentials.
    */

class RemoteCallbackHandler implements CallbackHandler {
   private String username;
   private String password;

   RemoteCallbackHandler(String username, String password){
         this.username = username;
         this.password = password;
   }
   public void handle(Callback[] cb) {
         for (int i = 0; i < cb.length; i++){
```

```
                if (cb[i] instanceof NameCallback){
                      NameCallback nc = (NameCallback)cb[i];
                      nc.setName(username);
                } else if (cb[i] instanceof PasswordCallback){
                      PasswordCallback pc = (PasswordCallback)cb[i];
                      pc.setPassword(password.toCharArray());
                      password = null;
                }
          }
    }
}
```

ServerProxy.java

```
package src.server;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import javax.security.auth.Subject;

import java.util.ArrayList;
import java.util.Date;

import src.permissionvalidation.*;
import src.common.ServerInterface;

/**
 * Proxy implements the same interface as the server.
 *
 * Protects the server from direct calls by clients. All calls by
 * clients have to pass security checks in the proxy.
 * Hides the implementation of the actual methods on the server.
 *
 * All the methods check is the caller has permissions, and only then
 * call methods for him. If the user doesnt have permission they throw
 * a SecurityException.
 */

public class ServerProxy extends UnicastRemoteObject implements ServerInterface {

   // A reference to the REAL server
   private /* spec_public */ ServerInterface theServer;
   // The particular user that is associated to this proxy
   private /* spec_public */ Subject theUser;

   //@ public ghost String state;

   // Constructor
   /*@ requires             true;
     @ ensures              subject!=null && theServer!=null;
     @ signals_only   RemoteException;
     @*/
   protected ServerProxy(Subject subject, ServerInterface theServer)
                                                         throws RemoteException {
         this.theServer = theServer;
         this.theUser = subject;
   }
   /*@ invariant (subject!=null && theServer!=null) &&
     @                   (state==VERIFY_AUTHORIZATION || state==CALL_METHOD);
     @*/

   /*@ constraint
     @          (state==VERIFY_AUTHORIZATION ==>
     @                      assert \old(state)==REQUEST_METHOD) &&
     @          (state==CALL_METHOD ==> \old(state)==VERIFY_AUTHORIZATION);
     @*/

   /*@   normal_behavior
```

```
  @          requires     cpr!=null && fName!=null && lName!=null &&
  @                                                adress!=null && DOB!=null;
  @    ensures            \only_called(checkPermission,insert);
  @
  @ also
  @      exceptional_behavior
  @           signals_only      RemoteException, SecurityException;
  @*/
public boolean insert(String cpr, String fName,
                   String lName, String adress, Date DOB)
                                 throws RemoteException, SecurityException {
      //@ set state==VERIFY_AUTHORIZATION;
      checkPermission("insert");
      //@ set state==CALL_METHOD;
      return theServer.insert(cpr, fName, lName, adress, DOB);
}

/*@    normal_behavior
  @          requires     cpr!=null;
  @    ensures            \only_called(checkPermission, findP) &&
  @                                    \result=!null && \fresh(\result);
  @ also
  @      exceptional_behavior
  @           signals_only      RemoteException, SecurityException;
  @*/
public String findP(String cpr)
                         throws RemoteException, SecurityException{
      //@ set state==VERIFY_AUTHORIZATION;
      checkPermission("findP");
      //@ set state==CALL_METHOD;
      return theServer.findP(cpr);
}

/*@    normal_behavior
  @          requires     cpr!=null;
  @    ensures            \only_called(checkPermission, editP) &&
  @                                                 \fresh(\result);
  @ also
  @      exceptional_behavior
  @           signals_only      RemoteException, SecurityException;
  @*/
public boolean editP(String cpr, String fName, String lName,
                         String adress) throws RemoteException,SecurityException {

      //@ set state==VERIFY_AUTHORIZATION;
      checkPermission("editP");
      //@ set state==CALL_METHOD;
      return theServer.editP(cpr, fName, lName, adress);
}

/*@    normal_behavior
  @          requires     cpr!=null && date!=null && test!=null &&
  @                                    comment!=null && totalCost!=null;
  @          ensures            \only_called(checkPermission,
  @                                    addMedicalRecords)&& \fresh(\result);
  @ also
  @    exceptional_behavior
```

```
      @           signals_only      RemoteException, SecurityException;
      @*/
    public boolean addMedicalRecords(String cpr, Date date,
                      ArrayList test, String comment, Double totalCost)
                                   throws RemoteException, SecurityException {

          //@ set state==VERIFY_AUTHORIZATION;
          checkPermission("addMedicalRecords");
          //@ set state==CALL_METHOD;
          return theServer.addMedicalRecords(cpr, date, test, comment, totalCost);
    }

    /*@   normal_behavior
      @          requires    cpr!=null;
      @          ensures            \only_called(checkPermission, listRecords) &&
      @                                                             \fresh(\result);
      @ also
      @   exceptional_behavior
      @           signals_only      RemoteException, SecurityException;
      @*/
    public ArrayList listRecords(String cpr)
                 throws RemoteException, SecurityException {

          //@ set state=VERIFY_AUTHORIZATION;
          checkPermission("listRecords");
          //@ set state=CALL_METHOD;
          return theServer.listRecords(cpr);
    }


    /**
       * Check if the current client can call a certain method.
       * The check is made through JAAS and its policy file.
       *
       * @param  methodName   The name of the method to verify if
       *                           the user has permission to run.
       */
    /*@ normal_behavior
      @          requires    methodName!=null && theUser!=null;
      @   assignable   \nothing;
      @          ensures            \only_called(doAs);
      @ also
      @     exceptional_behavior
      @           signals_only      SecurityException;
      @*/
    private void checkPermission(String methodName) throws SecurityException {
          Subject.doAs(theUser, new MethodcallValidate(methodName));
    }

}
```

ServerImpl.java

```
package src.server;

import java.io.*;
import java.rmi.RemoteException;
import java.util.*;

import src.common.ServerInterface;

  /*
   * The class where the use cases are actually implemented.
   */

public class ServerImpl implements ServerInterface {

   private Patient P;
   // The patient class
   private MedRecord M;
   // The Class containing a single medical record entry for a patient
   private ArrayList medicalRecords;
   // The list containing all medical records of a single Patient

   private Hashtable patientList;
   // Maps CPR-ids to the patient object

   private HashMap patientRecords;
   // Maps Patient CPR-ids to the ArrayList containing his records


   // CONSTRUCTOR
   public ServerImpl() {
         patientList = new Hashtable();
         patientRecords = new HashMap();
   }

   /**
    * Check if this patient ID exists, if it doesnt; insert a new patient
    * in the patient list
    */
   public boolean insert(String cpr, String fName,
                     String lName, String adress, Date DOB)
                           throws RemoteException, SecurityException {

         if (patientList.containsKey(cpr)==false) {
               // Patient does not exist. Insert it
               patientList.put(cpr, new Patient(fName, lName, adress, DOB));
               return true;
               } else {
                     return false;
                     // This Patient already exists. Do nothing.
               }

   }

   /**
```

```
 * Looks for the patient registered with this cpr ID.
 * If found it returns the patient data.
 * If NOT found it returns null.
 */
public String findP(String cpr) throws RemoteException, SecurityException {

	if (patientList.containsKey(cpr)) {
		P = (Patient) patientList.get(cpr);
		return P.toString();
	} else
		return null;

}
/**
 * Edits patients name, lastname or adress. Edits them only if the value
 * of their parameter is NOT an empty string. If it is an empty string
 * is given, then that field is not updated.
 */
public boolean editP(String cpr, String fName, String lName, String adress)
                                      throws RemoteException, SecurityException {

	// Check which of the fields has been provided by the caller
	// to be update that field
	if (patientList.containsKey(cpr)){
		P = (Patient) patientList.get(cpr);
			if (fName!="")
				P.setfName(fName);
			if(lName!="")
				P.setlName(lName);
			if(adress!="")
				P.setAdress(adress);

		// Place the newly updated object into the hashtable
		patientList.put(cpr, P);
		return true;
	}
// no user exists with this cpr ID.
return false;
}

/**
 * Check if patient exists. If not, no data is added.
 * If yes, check if he as previous records and append the newest record
 * to his list, if no he as previous records then create a record
 * list for him and append the newest record to his list.
 *
 */
public boolean addMedicalRecords(String cpr, Date date, ArrayList test,
                                String comment, Double totalCost)
                                              throws RemoteException, SecurityException
{

	if (patientList.containsKey(cpr)) {
	// This Patient exists in the list of patients

		// Check if he has a privious history
		if(patientRecords.containsKey(cpr)) {
```

```
                    // Patient has already a history of records
                    medicalRecords = (ArrayList) patientRecords.get(cpr);
                    // Get his list of medical records
                    M = new MedRecord(date, test, comment, totalCost);
                    // Create a new Record with the data recieved
                    // through parameters to add it to his previous records.
                    medicalRecords.add(M);
                    //Insert the newly created record in his list
                    return true;

            } else {
                    // This is the first record being entered for this patient
                    medicalRecords = new ArrayList();
                    // Create an (empty) history for him
                    M = new MedRecord(date, test, comment, totalCost);
                    // Create a new Record with the data recieved
                    // through parameters
                    medicalRecords.add(M);
                    //Insert the newly created record in his list
                    return true;
            }
        }
        return false;
        // This patient is not registered in our patients list thus,
        // no records can be added for him.
    }

    /**
     * List records of the patient with the given CPR ID number.
     * Return NULL if patient does not exist
     */
    public ArrayList listRecords(String cpr) throws RemoteException,SecurityException{
        // Check if this patient ID exists
        if (patientList.containsKey(cpr)) {
            // This Patient exists
                    medicalRecords = (ArrayList) patientRecords.get(cpr);
                    // Get the history of Patient

            return medicalRecords;
            // Return the ArrayList with history objects
        }
        return null;
        // Patient does not exist
    }

}
```

ServerInterface.java

```
package src.common;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.*;

/**
 * Server interface.  Containes the privileged
 * methods that require permissions to be executed.
 *
 * @throws SecurityException when the client
 *           doesn't have proper permissions to invoke the method.
 */

public interface ServerInterface extends Remote {


   public boolean insert(String cpr, String fName,
                      String lName, String adress, Date DOB)
                            throws RemoteException, SecurityException;


   public String findP(String cpr)
                            throws RemoteException, SecurityException;


   public boolean editP(String cpr, String fName,
                      String lName, String adress)
                            throws RemoteException, SecurityException;


   public boolean addMedicalRecords(String cpr, Date date,
                      ArrayList test, String comment, Double totalCost)
                                        throws RemoteException, SecurityException;


   public ArrayList listRecords(String cpr)
                      throws RemoteException, SecurityException;


}
```

LoginModule.java

```
package src.server;

import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.LoginException;
import java.util.*;

/**
 * Extends the built-in class of JAAS that makes the yes/no decission
 * related to the user authentication.
 * LoginModule authenticates users from credentials, and adds the
 * permissions that were listed for that user in the policy.
 */

public class LoginModule implements javax.security.auth.spi.LoginModule {

   private boolean debug = false;

   private Subject subject;
   private MyPrincipal entity;
   private CallbackHandler callbackHandler;
   private Map sharedState;
   private Map options;

   /* Table holds credentials of the registered users.
    * It holds their usernames and their passwords.
    */
   private Hashtable userList;

   // tracking the authentication status
   private static final int NOT = 0, OK = 1, COMMIT = 2;
   private int status;

   // current user
   private String username;
   private char[] password;

   //CONSTRUCTOR
   protected LoginModule() {
         userList = new Hashtable();

         // Username & passwords of registered users in the clinic.
         // Inserted for testing of the application.
         userList.put("admin1", "admin1");
         userList.put("admin2", "admin2");
         // users with Administration priveleges
         userList.put("med1", "med2");
         userList.put("med2", "med2");
         // users with Medical staff priveleges
   }

   /*
    * This method is called if the LoginContext's overall authentication
    * failed. Cleans up any state if any of modules succeded but the overal
```

```
 * authentication doesnt succed.
 */
public boolean abort() throws LoginException {
      if(status == NOT) {
          return false;
      } else if( status == OK ) {
          // login succeeded but overall authentication failed
          username = null;
          if( password != null ) password = null;
          entity = null;
      } else {
          // overall authentication succeeded and commit succeeded,
          // but someone else's commit failed
          logout();
      }//end if/else
      status = NOT;
      return true;
}

/*
 * Called if the overall authentication succeeds. Associates a
  * MyPrincipal object with the Subject located in the LoginModule.
 */
public boolean commit() throws LoginException {
      if(status == NOT || subject == null) {
          return false;
      } else {
          // add a Principal (authenticated identity) to the Subject

          // assume the user we authenticated is the MyPrincipal
          entity = new MyPrincipal(username);
          Set entities = subject.getPrincipals();
          if( !entities.contains(entity) )
                  entities.add(entity);

          // in any case, clean out state
          username = null;
          password = null;

          status = COMMIT;
          return true;
      }
}

/**
  * Initialize the LoginModule
  */
public void initialize(Subject subject, CallbackHandler cbH,
                                                Map sharedState, Map options) {

      status = NOT;
      this.subject = subject;
      this.callbackHandler = cbH;
      this.sharedState = sharedState;
      this.options = options;

}
```

```
/**
  * Authenticate the user based on provided credentials.
  * If the credential matches the expected one,
  * authentication succeeds - otherwise it doesnt.
  */
public boolean login() throws LoginException {
      if( callbackHandler == null )
          throw new LoginException( "Error: no CallbackHandler " +
                              "available to retrieve user credentials");

      Callback[] callbacks = new Callback[2];
      callbacks[0] = new NameCallback( "\nuser name: " );
      callbacks[1] = new PasswordCallback( "password: ", false );

      try {
            callbackHandler.handle(callbacks);
            //get the user credentials
          username = ((NameCallback)callbacks[0]).getName();

          char[] tmpPassword =
            ((PasswordCallback)callbacks[1]).getPassword();

          if (tmpPassword == null)
            // treat a NULL password as an empty password

            tmpPassword = new char[0];
          password = new char[tmpPassword.length];

          System.arraycopy( tmpPassword, 0,password, 0,
                                          tmpPassword.length );
          ((PasswordCallback)callbacks[1]).clearPassword();
          //wipe out occurrences in memory
      }
      catch( java.io.IOException ioe ) {
          throw new LoginException(ioe.toString());
      }
      catch( UnsupportedCallbackException uce ) {
          throw new LoginException( "Error: " +
                         uce.getCallback().toString() +
                              " not available to authenticate user." );
      }


      // Lookup the given credentials in our HashMap to see if they're
      // legitimate users of the clinic
      if (lookupUser(username, password))
            // because the given username and passwords is
            // found in list, authentication succeeded!
            return true;
            else
                  // authentication failed
                  return false;
}

/**
 * Check if username and password exist in the userList HashTable.
```

```
 *
 * @param    usr         username provided by the client
 * @param    pwd         password provided by the client
 * @return   true   if username and password was correct
 *                  false if username and/or password incorrect
 */
private boolean lookupUser(String usr, char[] pwd) {

      if(userList.containsKey(usr)) {
            String key = (String) userList.get(usr);
            // Retrive that username
            if((userList.get(key)).toString().toCharArray()==pwd) {
                  // The password maches.
                  // This is a valid user.
                  key = null;//Clean up the String location in the memory
                                 //containing the password of the user;
                  return true;
            }
      }
      return false;
      // Username does not exist
}


 /**
  * Logout the user.
  * Removes the MyPrincipal added by the .commit() method.
 */

public boolean logout() throws LoginException {

      subject.getPrincipals().remove(entity);
      status = NOT;
      username = null;

      if( password != null ) password = null;
      entity = null;
      return true;
}

}
```

LoginInterface.java

```
package src.common;

import java.rmi.Remote;
import javax.security.auth.login.LoginException;

/**
 * The interface of the login object to which (remote) users connect
 * to authenticate.
 */

public interface LoginInterface extends Remote {

   /**
    * Remote method that forwards the credentials from the
    * client software to the server for user authentication.
    */
   public ServerInterface forwardCrds(String username, String password)
                                throws java.rmi.RemoteException, LoginException;

}
```

MethodcallValidate.java

```
package src.permissionvalidation;
import src.permissions.ServerPermission;

import java.security.AccessController;
import java.security.PrivilegedAction;

/*
 *  Test if the user has the neccessary permissions to call
 *  the method in the login context of his authentication.
 * For verifying the availability of the permission,
 * a ServerPermission object is required.
 */
public class MethodcallValidate implements PrivilegedAction {

   // Name of the method to be checked
   private String priveledgedMethodName;

   public MethodcallValidate (String method) {
         priveledgedMethodName = method;
   }

   public Object run() {
         // Check if the appropriate ServerPermission is owned by
       // the user. If not an exception is thrown.

         AccessController.checkPermission(new
                                 ServerPermission(priveledgedMethodName));
         return null;
   }

}
```

ServerPermission.java

```
package src.permissions;
import java.security.BasicPermission;

/*
 * Class for creating permission objects attached to the Subject class
 * when the the user is authenticated successfully.
 */

public class ServerPermission extends BasicPermission {

   public ServerPermission(String name) {
         super(name);
   }

   public ServerPermission(String name, String actions) {
         super(name, actions);
   }

}
```

MyPrincipal.java

```
package src.server;

import java.io.Serializable;
import java.security.Principal;

 /*
  * The implementation of the java.security.Principal interface that adds
  * an identity to the user.
  * To this user identity permission objects are added based on
  * the privileges that the administration has assigned to the user
  * through the policy file.
  *
  */
public class MyPrincipal implements Principal, Serializable {

   private String name;
   // Name of the principal

   // Constructor
   public MyPrincipal(String n) {
         name = n;
   }

   public String getName() {
         return name;
   }


   public String toString() {
         return( "MyPrincipal:  " + name );
   }

   public boolean equals( Object o ) {
         if( o == null )
             return false;
         if( this == o )
             return true;

         if( !(o instanceof MyPrincipal) )
             return false;
         MyPrincipal that = (MyPrincipal) o;

         if( this.getName().equals(that.getName()) )
             return true;
         return false;
    }


   public int hashCode() {
         return name.hashCode();
   }
}
```

MedRecord.java

```
package src.server;

import java.io.Serializable;
import java.util.*;

 /*
  * The medical record contains an ArrayList of test (objects),
  * the test date, related coments and the cost of the tests during
  * the visit to the clinic.
  */
public class MedRecord implements Serializable {

    private Date date;
    // Date when the (medical) test was done
    private ArrayList tests;
    // Test objects
    private String comment;
    private Double totalCost;

    public Iterator it;

    // Constructor
    public MedRecord(Date d, ArrayList t, String c, Double tc) {
            date = d;
            tests = t;
            comment = c;
            totalCost = tc;
    }

    /**
     * Return a string representation of a Medical record object
     */
    public String toString() {
            String allTests = "";
            // First, get a string representation of all Test objects
            for (int index = 0; index <= tests.size(); index++) {
                    Test t = (Test) tests.get(index);
                    allTests = allTests + t.toString() + "\n";
            }

            // Join the other records with the tests string
            String record = date.toString() + " " + allTests +
                                    " " + comment + " " + totalCost.toString() + "\n";
            return record;
    }

}
```

Patient.java

```
package src.server;

import java.util.Date;
   /*
    * The patient class, for holding all the Patient data.
    */
public class Patient {
   String fName;
   // First name of the patient
   String lName;
   // Last name of the patient
   String adress;
   // Adress of the patient
   Date DOB;

   // Constructor of the patient class
   public Patient(String fName, String lName, String adress, Date DOB) {
         this.fName = fName;
         this.lName = lName;
         this.adress = adress;
         this.DOB = DOB;
   }

   // Getter methods to retrieve values of the fields in the patient object
   public String getfName() {
         return fName;
   }

   public String getlName() {
         return lName;
   }

   public String getAdress() {
         return adress;
   }


   // Setter methods to update values of the fields in the patient object
   public void setfName(String fn) {
         fName = fn;
   }

   public void setlName(String ln) {
         fName = ln;
   }

   public void setAdress(String ad) {
         adress = ad;
   }

   public String toString() {
         return fName + " " + lName + "\n " + adress + " " + "\n " +DOB;
   }
}
```

Test.java

```
package src.server;

import java.io.Serializable;

 /*
  * Medical Tests done on the patient.
  */
public class Test implements Serializable {

   private String name;
   private String result;

   public Test(String n, String t) {
          setName(n);
          setResult(t);
   }

   // Getter and setter methods for name & result fields
   public void setName(String name) {
          this.name = name;
   }

   public String getName() {
          return name;
   }

   public void setResult(String result) {
          this.result = result;
   }

   public String getResult() {
          return result;
   }

   public String toString() {
          return name + " " + result + " ";
   }
}
```

Server.java

```
package src.server;

  /**
   * The main class for the server.
   */
import java.rmi.registry.*;
import java.util.Date;
import src.common.*;

public class Server {


   public static void main(String[] args) {
         /* Ensures that the identifiers generated for the server objects
        * will be secure
        */
         System.setProperty("java.rmi.server.randomID", "true");

         try {
                ServerInterface theServer = new ServerImpl();
                LoginInterface loginObject = new LoginImpl(theServer);

                Registry loginRegistry = LocateRegistry.createRegistry(6000);
                //Creates and exports a Registry on the local host
                // that accepts requests on the specified port.
                loginRegistry.bind("RemoteLoginServer", loginObject);
                // Binds a remote reference of the specified object name
                // in the registry.

                System.out.println(new Date() + ": Server up and running");
                // Print a message to know that the server is operational

         } catch (Exception e) {
               e.printStackTrace();
       }


   }

}
```

Appendix C

F. Policy and configuration files

Authorizations.policy

```
   // The Authorization policy file granting executions priveleges
   // to each user seperatelly, for invoking methods that complete
   // the use cases.

   //    ADMINISTRATION users

grant Principal src.server.MyPrincipal "admin1"
{
   permission src.permissions.ServerPermission "insert";
   permission src.permissions.ServerPermission "find";
   permission src.permissions.ServerPermission "editP";
};

grant Principal src.server.MyPrincipal "admin2"
{
   permission src.permissions.ServerPermission "insert";
   permission src.permissions.ServerPermission "find";
   permission src.permissions.ServerPermission "editP";
};

   //    MEDICAL STAFF users

grant Principal src.server.MyPrincipal "med1"
{
   permission src.permissions.ServerPermission "addMedicalRecords";
   permission src.permissions.ServerPermission "listRecords";
};

grant Principal src.server.MyPrincipal "med2"
{
   permission src.permissions.ServerPermission "addMedicalRecords";
   permission src.permissions.ServerPermission "listRecords";  };
```

LoginModule.config

```
   // Name of the login module to initiate.

JAASLogin {
   src.server.LoginModule required;
};
```

codePermissions.policy

```
   // The list of permissions assigned to to JAR archives about the
   // system resources thay can use.

grant codebase "file:server.jar"
{

     permission java.util.PropertyPermission
                                      "java.rmi.server.randomID", "write";

     permission java.net.SocketPermission "127.0.0.1:1024-","accept";

     permission javax.security.auth.AuthPermission
                                      "createLoginContext.JAASLogin";

     permission javax.security.auth.AuthPermission "modifyPrincipals";

     permission javax.security.auth.AuthPermission "addMedicalRecords";
     permission javax.security.auth.AuthPermission "editP";
     permission javax.security.auth.AuthPermission "findP";
           permission javax.security.auth.AuthPermission "insert";
           permission javax.security.auth.AuthPermission "listRecords";
                     // Names of the permissions that the users can have
                     // when a login context is created for them.

     permission src.permissions.ServerPermission "*";

};

grant codebase "file:actions.jar"
{
};
```

Bibliography

1. **Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte.** The Spec# programming system: An overview. *In Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS).* 2004, Vol. 3362, Springer.
2. **Pugh., D. Hovemeyer and W.** *Finding bugs is easy.* s.l. : ACM Press, 2004.
3. **Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata.** Extended static checking for Java. *In PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation.* ACM Press., 2002.
4. **James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Roby.** *Bandera: source-level interface for model checking Java programs.* s.l. : ACM Press, 2000.
5. **Pnueli., Z. Manna and A.** *The Temporal Logic of Reactive and Concurrent System: Specification.* s.l. : Springer-Verlag, 1991.
6. **L., Requet. A and Burdy.** Jack: Java Applet Correctness Kit. *In Gemplus Developer Conference.* 2002.
7. **Jacobs, Joachim van den Berg and Bart.** *The LOOP Compiler for Java and JML.* London, UK : Springer-Verlag, 2001.
8. **Mariela Pavlova, Mariela Pavlova, Gilles Barthe, Gilles Barthe, Lilian Burdy, Lilian Burdy , Marieke Huisman , Marieke Huisman, Jean-louis Lanet, Jean-louis Lanet, Thme Gnie Logiciel.** *Enforcing High-Level Security Properties for Applets.* Sophia Antipolis, France : INRIA, 2004 .
9. *Source code verification of a secure payment applet.* **Bart Jacobs, Martijn Oostdijk and Martijn Warnier.** Issues 1-2, Nijmegen, The Netherlands. : Elsevie, 2004, Vol. Volume 58.
10. **Umberto Costa, Anamaria Moreira, Martin Musicante, and Placido Souza Neto.** *Specification and Runtime Verification of Java Card Programs.* Salvador, Brazil : s.n., 2008.
11. **Rischpater, Ray.** *Beginning Java ME Platform.* s.l. : Apress, 2008. 1430210613.
12. *Implementing a Formally Verifiable Security Protocol in Java Card.* **Engelbert Hubbers, Martijn Oostdijk, and Erik Poll.** Berlin : Springer Verlag.
13. **Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino and Erik Poll.** *An overview of JML tools and applications.* Berlin : Springer, 2004.

14. **Aleksy Schubert, Jacek Chrzaszcz.** ESC/Java2 as a Tool to Ensure Security in the Source Code of Java Applications. *Software Engineering Techniques: Design for Quality.* Boston : Springer, 2007.
15. *Source code verification of a secure payment applet.* **Bart Jacobs Martijn Oostdijk, Martijn Oostdijk and Martijn Warnier.** Issues 1-2, Nijmegen, The Netherlands. : Elsevie, 2004, Vol. Volume 58.
16. **Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman and Jean-Louis Lanet.** *Enforcing High-Level Security Properties for Applets.* s.l. : INRIA Sophia Antipolis, France and INRIA Dir DRI, France, 2003.
17. **Warnier, Martijn Oostdijk and Martijn.** *On the combination of Java Card Remote Method Invocation and JML.* Nijmegen, The Netherlards : Dept. Computer Sci., Univ. Nijmegen, 2003.
18. **Jacobs., J. van den Berg and B.** The LOOP compiler for Java and JML. [book auth.] T. Margaria and W. Yi. *Tools and algorithms for the construction and Analysis of Systems.* Berlin : Springer, 2002.
19. **Erik Poll, Aleksy Schubert.** Verifying an implementation of SSH. *Proceedings of the 17th Workshop on Information Technology and Systems (WITS'07).* Montreal, Canada : Concordia University, 2007.
20. **Engelbert Hubbers, Martijn Oostdijk, and Erik Poll.** Implementing a Formally Verifiable Security Protocol in Java Card. *Security in Pervasive Computing.* Berlin, Heidelberg : Springer, 2004.
21. **J. Clark, J. Jacob.** *A Survey of Authentication Protocol Literature: Version 1.0.* 1997.
22. **Tamalet., Marieke Huisman and Alejandro.** *A Formal Connection between Security Automata and JML Annotations.* Berlin, Heidelberg : Springer-Verlag, 2009.
23. **Oostdijk, Martijn Warnier and Martijn.** *Non-interference in JML.* Nijmegen, The Netherlands : Nijmegen Institute for Computing and Information Sciences, 2005. ICIS-R05034.
24. **Leino, R.Joshi K.** A semantic approach to secure information flow. *Science of Computer Programmimg.* 2000, Vols. 37(1-3), 113–138.
25. **Perumandla, Yoonsik Cheon and Ashaveena.** *Specifying and Checking Method Call Sequences in JML.* s.l. : CSREA Press., February 2005.
26. **Cheo, Gary T. Leavens and Yoonsik.** *Design by Contract with JML.* August 17, 2005.
27. **Cheon, Yoonsik.** *A runtime assertion checker for the Java Modeling Language.* s.l. : Iowa State University, Department of Computer Science., April 2003.
28. **Gary T. Leavens, Yoosnik Cheon, Curtis Clifton, Clyde Ruby and David Cok.** *How the design of JML accommodates both runtime assertion checking.*
29. **Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R.** *JMLReference Manual.* s.l. : Available from http://www.jmlspecs.org, October 2007.
30. **Gary T. Leavens, Albert L. Baker, and Clyde Ruby.** *Preliminary design of JML: a behavioral interface specification language for Java.* s.l. : SIGSOFT Softw. Eng. Notes, 31(3):1–38, 2005.

31. **Patrice Chalin, Joseph R. Kinry, Gary Leavens and Erik Poll.** Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. *Formal Methods for Components and Objects.* s.l. : Springer, 2006.
32. **Meyer, Bertrand.** *Object-Oriented Software Construction, Second.* s.l. : Prentice Hall, 1997.
33. **Gary T. LEavens, Albert L. Baker and Clyde Ruby.** *JML: A Java Modeling Language.* Ames, Iowa, USA. : Department of Computer Science, Iowa State University., 1998.
34. **Cay S. Horstmann, Gary Cornell.** *Core Java™ 2 Volume II - Advanced Features, Seventh Edition.* s.l. : Prentice Hall, November 22, 2004. 0-13-111826-9.
35. **Ganguli, Harpreet.** *Java Security.* Cincinati, Ohio 45208, USA : Premier Press, 2002. 1-931841-85-3.
36. **Rubin, James.** *Java security: Authentication and authorization.* s.l. : IBM Corporation., 2002.
37. **Nestor Catatano, Tim Wahls.** *Executing JML Specifications of Java Card Applications: A Case Study.* Hawaii : In Proceedings of the ACM Symposium on Applied Computing, Software Engineering Track (SAC-SE), March, 2009.
38. **Mariela Pavlova, Gilles Barthe, Gilles Barthe, Lilian Burdy, Lilian Burdy , Marieke Huisman , Marieke Huisman, Jean-louis Lanet, Jean-louis Lanet, Thme Gnie Logiciel.** *Enforcing High-Level Security Properties for Applets.* Sophia Antipolis, France : INRIA, 2004.
39. **Christian Haack, Erik Poll, Aleksy Schubert.** *Explicit information flow properties in JML.* 2008.
40. *The Daikon system for dynamic detection of likely invariants.* **Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao.** 1--3, s.l. : Science of Computer Programming, 2007, Vol. 69.
41. **B. Jacobs, C. March and N. Rauch.** Formal verification of a commercial smart card applet with multiple tools. [book auth.] S. Maharaj and C. Shankland C. Rattray. *Algebraic Methodology and Software Technology.* Berlin : Springer, 2004.
42. **Warnier, Martijn.** *Language Based Security for Java and JML, Phd thesis.* Nijmegen, The Netherlands : Radboud University, 2006. 90-9020922-0.
43. **Valmari, Antti.** The state explosion problem. [book auth.] Grzegorz Rozenberg Wolfgang Reisig. *Lectures on Petri nets I: basic models : advances in petri nets.* 1998.
44. **Myers, A. Sabelfeld & A.C.** *Language-Based Information-Flow Security.* s.l. : IEEE Journal on selected areas in communications, 2003.
45. **Meseguer, J. Goguen and J.** *Security policies and security models.* s.l. : IEEE Symp. on Security and Privacy, pp. 11–20, IEEE Comp. Soc. Press, 1982.
46. *A Type-Based Approach to Program Security.* **Smith, D. Volpano and G.** s.l. : In Proc. 7th International Joint Conference on the Theory and Practice of Software Development, 1997, Vol. 1214.
47. **Peter Rob, Carlos Coronel, Keeley Crockett.** *Database Systems: Design, Implementation & Management.* London : Thomson Learning, 2008. 978-1844807321.

48. **Yao, D. Dolev and A.C.** On the security of public key protocols. *IEEE - Annual Symposium on Foundations of Computer Science.* 22nd, 1981, Vols. pp. 350-357.

49. **Watson, Carl.** *Beginning C# 2005 databases.* 2006. ISBN 978-0-470-04406-3.

50. **C. Farkas, T. Toland, C. Eastman.** *The Inference Problem and Updates in Relational Databases.* s.l. : Working Conference on Database and Application Security, 2001.

51. **Kose, Ilker.** Distributed Database security. *GYTE, Computer Engineering.* 2002.

52. **Kumar, Pankaj.** *J2EE Security for Servlets, EJBs, and Web Services.* s.l. : Prentice Hall, 2003. 0131402641.

53. **Wagner, F.** *Modeling Software with Finite State Machines: A Practical Approach.* s.l. : Auerbach Publications, 2006. 0-8493-8086-3.

54. **Breunesse, E. Poll & C. B.** Verifying JML specifications with model fields. *In formal techniques for Java-like programs.* Proceeding of the ECOOP, 2003 Workshop., 2003.

55. **Gary T. Leavens, Albert Baker, Clyde Ruby.** JML: A Nnotation for Detailed Design. [book auth.] Berhard Rumpe and William Harvey. Haim Kilov. *Behavioral Specification for Business Systems, chapter 12.* s.l. : Kluver Academic Publishers, 1999.

56. **Yoosnik Cheon, Gary T. Leavens, Murali, Sitaraman, and Stephen Edwards.** *Model variables: Cleanly supporting abstraction in design by contract.* May 2005. 35(6):583-599.

57. **Hoare, C. A. R.** *An axiomatic basis for computer programming.* s.l. : Communications of the ACM, 1969. 12(10):576-583.

58. **Cunningham, R Mugridge & W.** *Fit for Developing Software: Framework for Integrated Tests.* s.l. : Prentice Hall PTR, 2005. 0-321-63049-1.

59. **Murata, Tadao.** *Petri Nets: Properties, Analysis and Applications.* 1989.