Software Engineering for Secure Systems:

Industrial and Research Perspectives

Haralambos Mouratidis
University of East London, UK

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE

Hershey · New York

Director of Editorial Content: Kristin Klinger
Director of Book Publications: Julia Mosemann
Acquisitions Editor: Lindsay Johnston
Development Editor: Christine Bufton
Typesetter: Michael Brehm
Production Editor: Jamie Snavely
Cover Design: Lisa Tosheff

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Software engineering for secure systems : industrial and research perspectives / Haralambos Mouratidis, editor.
p. cm.
Includes bibliographical references and index.
Summary: "This book provides coverage of recent advances in the area of secure software engineering that address the various stages of the development process from requirements to design to testing to implementation"--Provided by publisher.
ISBN 978-1-61520-837-1 (hardcover) -- ISBN 978-1-61520-838-8 (ebook) 1. Computer security. 2. Software engineering. I. Mouratidis, Haralambos, 1977-
QA76.9.A25S6537 2010
005.8--dc22
2010017214

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 9
Privacy Aware Systems:
From Models to Patterns

Alberto Coen-Porisini
Università degli studi dell'Insubria, Italy

Pietro Colombo
Università degli studi dell'Insubria, Italy

Sabrina Sicari
Università degli studi dell'Insubria, Italy

ABSTRACT

Enterprises have adopted various strategies to protect customers' privacy and to make public their policies. This chapter presents a conceptual model for supporting the definition of privacy policies. The model, described by means of UML, introduces a set of concepts concerning privacy and defines the existent relationships among those concepts along with the interfaces for the definition of privacy related mechanisms. The chapter also illustrates how the conceptual model can be used to build design solutions for three recurrent requirements for privacy aware systems concerning the definition of anonymity, the acquisition of the informed consent, and privacy policies enforcement. The proposed problems are separately illustrated and a solution based on the conceptual model is described for each of them. Finally, in order to assess the model and the design solutions, this chapter presents an example concerning the health domain.

INTRODUCTION

Nowadays privacy is a key issue and has received increasing attention from consumers, companies, researchers and legislators. Legislative acts, such as the European Union Directive[1] for personal data, the Health Insurance Portability and Accountability Act[2] for healthcare and the Gramm Leach Bliley Act[3] for financial institutions, require governments and enterprises to protect the privacy of their citizens and customers, respectively. Although enterprises have adopted various strategies to protect customers privacy and to make public their privacy policies (e.g., publishing a privacy policy on web-sites possibly based on P3P[4]), none of these approaches include systematic mechanisms to describe how personal data are actually handled after they are collected.

This chapter proposes a conceptual model that provides a sound foundation for the definition of

DOI: 10.4018/978-1-61520-837-1.ch009

privacy policies. The model, which extends the work proposed by Coen-Porisini & al. (2007), is defined using UML[5] and represents a general schema that can be easily adopted in different contexts.

A privacy policy defines the way in which data referring to individuals can be collected, processed and diffused according to the rights that individuals are entitled to. Thus, the model introduces the concepts, such as users, data, actions, that are needed in order to define a privacy policy along with the existing relationships among them.

Although the model introduces all the elements that are required for the definition of privacy aware systems, it operates at a conceptual level with a very high level of abstraction. The main benefit of this approach is represented by the fact that the model is domain independent and it can be used in different contexts. In this way analysts and designers can describe privacy related features/ requirements and then they can integrate them at design time in new or existing systems exploiting the visibility and usability of UML.

In addition to presenting the above mentioned model, this chapter introduces a design solution to some privacy related requirements that are common to most privacy aware systems. The way in which such design solutions are provided is by means of design patterns (Gamma et al. 1994), which constitute a set of design guidelines and schemes that can drive the designer towards the specification of a privacy aware system.

In this chapter, for space reasons, we focus on the following three requirements: anonymity, informed consent acquisition and privacy policy enforcement. Notice that other privacy related requirements such as pseudonymity, unobservability and so on can be addressed in the same way by developing appropriate design patterns.

Anonymity is an important requirement for a privacy aware system that aims at protecting the identity of the individuals whose data are handled by the system. In general, data can be categorized into different classes. Among them, one class includes data, referred to as *sensitive data*, concerning the private life, political or religious creed and so on, while another class contains data that describes the identity of individuals (e.g., first name, family name, etc.). A privacy aware system must assure that only authorized users can view the existing relationship between sensitive data and the identity of the individuals.

Informed consent is another important requirement for privacy aware systems that aims at assuring individuals that the system will use their data according to their will. For instance many legislations require that individuals must be informed of both the reasons for which the system will handle their data and the way in which data processing is performed. In such cases every individual has to provide an explicit consent before any data processing can occur.

Privacy policies enforcement requires that the activities performed within a system are checked against the privacy policy in order to avoid any privacy violation.

Finally, in order to test the effectiveness of the conceptual model and of the proposed design solutions, we discuss their application by means of an example concerning the healthcare domain.

In the last few years, hospitals, clinics, surgeries, and diagnostic centers have increasingly adopted Information Technology-supported healthcare solutions in order to manage health-related information and to provide a (semi)automated administration of clinical functions. As a consequence, due to its critical nature, the healthcare domain represents an ideal field for experimenting the definition of privacy mechanisms.

The rest of the chapter is organized in the following way: Section 2 provides an overview of the main related works concerning privacy; Section 3 introduces the privacy model and discusses its main features; Section 4 illustrates how the proposed model can be used for defining design solutions that achieve specific requirements such as the anonymity, the informed consent and the enforcement of privacy policies; Section 5 presents

an application scenario in the healthcare domain; finally, Section 6 draws some conclusions.

BACKGROUND

While research on security is a well-established field, the issues that arise when dealing with privacy have been under thorough investigation only in the recent years. The research efforts aiming at the protection of individuals privacy can be partitioned in two main categories: Security-oriented Requirement Engineering (SRE) methodologies and Privacy Enhancing Technologies (PETs).

The former focuses on methods for taking into account security and privacy issues during the early stages of systems development, while the latter describes techniques to ensure privacy.

Several existing requirement engineering methodologies, such as Kaos (Lamsweerde & al. 2000), Tropos (Liu & al., 2002), Secure Tropos (Mouratidis & al., 2003a; Mouratidis & al., 2003b; Mouratidis & Giorgini, 2007), NFR (Chung, 1993; Mylopolulos & al., 1992) and GBRAM (Anton, 1996), can be used to take into account security issues at design level.

All the above methodologies address the problem of how to state as clearly as possible the requirements that an information system must satisfy in order to be considered secure (with respect to a set of given security policies). This is different from our goal, which is to define a conceptual model for representing privacy policies.

Kalloniatis & al. (2008) present a methodology, called PRIS, to incorporate privacy requirements into the system design process. PRIS is a requirement engineering methodology focused on privacy issues rather than on security requirements although it can be applied to the latter as well. It is based on the Enterprise Knowledge Development (EKD) framework, which is a systematic approach for developing and documenting organisational knowledge.

PRIS considers privacy requirements as organisational goals that need to be satisfied and adopts the use of privacy-process patterns as a way to: (1) analyse the impact of privacy requirement(s) on organisational goals, sub-goals and processes; and (2) facilitate the identification of the best system architecture supporting privacy-related business processes.

Thus, PRIS provides a complete view of the system including both the enterprise and privacy goals and refines the latter to identify a set of privacy requirements.

Instead, our approach introduces a set of concepts concerning privacy such as users, data, actions, and it defines the existent relationships among them, providing in this way a high level conceptual model, described in UML, that can be used to model privacy policies, which can be used to satisfy specific privacy requirements. In fact, in our approach, privacy requirements are addressed by introducing design patterns derived from the conceptual model. More specifically, our design patterns represent a set of design guidelines and schemes that can drive the designer towards the specification of a privacy aware system. In this way analysts and designers can describe privacy related features/requirements and then they can integrate them at design time in new or existing systems exploiting the visibility and usability of UML.

Agrawal & al. (2005) provide extensions to a RBDMS in order to express P3P privacy policies, at schema definition level. Furthermore, the authors define mechanisms for translating P3P privacy policies into a properly extended SQL-like data definition language. This is different from our approach, since what we propose is a conceptual model for the definition of privacy policies (not necessarily expressed in P3P) and for the specification of the needed functional modules of an application in order to enforce such policies.

Finally, in the field of SRE methodologies, several techniques have been proposed in order to

protect private data from unauthorized accesses. Typical examples are anonymizing techniques based on data suppression or randomization (Mielikinen, 2004; Narayanan & Shmatikov, 2005). However, these techniques do not require the definition of any privacy policies; rather they can be used as building blocks for realizing them.

The literature also reports many works that propose patterns and design guidelines for addressing the requirements imposed by specific security and privacy problems.

Many security patterns were defined to address enterprise, architectural and user-level security (Yoder & al. 1997; Blakley & al. 2004; Chung & al., 2004; Steel & al., 2005; Schumacher & al., 2006), while, presently, only few contributions concerning privacy have been defined. Chung & al. (2004) define privacy patterns for ubiquitous computing domain.

Schummer (2004) describes the privacy masquerade pattern, i.e., a pattern that specifies how it is possible to prevent personal information from being improperly transmitted.

Schumacher (2002) describes two privacy patterns, named Pseudonymous Email and Protection against Cookies, respectively. The former specifies mechanisms for hiding the sender of an email message; while the latter describes how to control the cookies in a web browser. Romanosky & al. (2006) introduce privacy patterns for online interactions, distinguishing between patterns for system architecture issues and patterns for end-user support. Hafiz (2006) defines anonymity design patterns for various types of online communication systems, online data sharing, location monitoring, voting and electronic cash management.

All the just mentioned patterns, however, address specific application domain issues, while the solution that we propose in the following sections is more general and can be applied to different contexts.

MODELLING PRIVACY

In order to model privacy policies it is necessary to introduce concepts such as users, data, actions and so on. The rest of the chapter adopts the terminology introduced by the EU directive, which is summarized in what follows:

- *personal data* means any information relating to an identified or identifiable natural person (referred to as data subject or subject).
- *processing of personal data (processing)* means any operation or set of operations which is performed upon personal data, whether or not by automatic means, such as collection, recording, organization, storage, adaptation or alteration, retrieval, consultation, use, disclosure by transmission, dissemination or otherwise making available, alignment or combination, blocking, erasure or destruction;
- *controller* means the natural or legal person, public authority, agency or any other body which alone or jointly with others determines the purposes and means of the processing of personal data;
- *processor* means a natural or legal person, public authority, agency or any other body which processes personal data on behalf of the controller;
- *the data subject's consent (consent)* means any freely given specific and informed indication of his/her wishes by which the data subject signifies his/her agreement to personal data relating to him/her being processed.

Moreover, as a distinctive feature of a privacy policy, the processor is allowed to execute given processing actions only under explicit *purposes* and *obligations*. A purpose describes for what aims data are processed, and it can be defined either as a high-level activity (e.g., "marketing", "customer

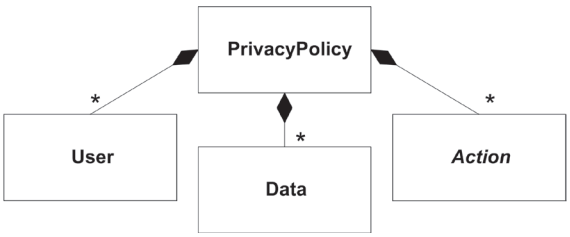
satisfaction") or as a set of actions (e.g., "compute the average price", "evaluate the customer needs"). An obligation is a set of actions that the processor guarantees to perform, after the data have been processed, that is after the execution of processing actions. Controllers define processing actions, as well as purposes and obligations and are required to verify that the former are executed according to the latter.

Subjects, whenever their data are collected, must grant their consent before any processing can be done and must be informed of the purposes and of the obligations related to any processing. Notice that, the consent can be given selectively that is, a subject can grant the consent for one purpose, while denying it for another one.

Starting from the previously presented terms we introduce several concepts and refinement. More specifically, data handled by a system are categorized into different classes. Among them, one class includes *sensitive data*, that is data concerning the private life, political or religious creed, health conditions and so on. Another class contains *identifiable data*, that is, data describing the identity of individuals (e.g., first name, family name, address, telephone, etc.).

Finally, we define the concepts of *role* and *function*. The role (Ni & al., 2007) specifies whether an individual is a subject, a controller or a processor, while the function represents the task performed by an individual within an organization. Thus, role is a cross cutting concept that is domain independent, while function strictly depends on the application domain. For example, in the context of a hospital information system we may have different functions, such as doctor, nurse, head-nurse, employee, and so on. Notice that, a function implicitly defines the set of actions that can be executed by an individual. For instance, a doctor is allowed to prescribe therapies and access patients' case histories, while an employee is allowed to make an appointment for a medical examination.

Figure 1. The privacy policy class diagram



Therefore, given an application scenario, each individual is characterized by a pair function-role, which specifies his/her behavioral profile with respect to a privacy policy.

THE UML MODEL

In the following we introduce a UML model that specifies the concepts occurring in a privacy policy along with their relationships. First of all we describe the static aspects of the model, by introducing all the structural elements involved by means of Class diagrams. Then, we describe by means of several Sequence diagrams the behavioral aspects of the model by specifying the basic interactions occurring among the previously introduced structural elements.

Figure 1 depicts a class diagram that provides a high level view of the basic structural elements of the model.

A privacy policy is represented by means of class *PrivacyPolicy*, which is composed of three different classes named *User*, *Data* and *Action*, respectively. Thus, an instance of *PrivacyPolicy* is characterized by specific instances of *User*, *Data* and *Action*.

Let us focus on the classes introduced by the diagram:

- *User* represents an actor either interested in processing data or involved by such a processing. Users are characterized by functions and roles. More specifically *Function*

represents the employment of a user in an application domain, while *Role* characterizes users with respect to privacy. As a consequence, *Role* is extended by three distinct classes to represent the different roles: *Subject*, which is anyone whose data are referred to, *Processor*, which is anyone who asks for processing data by performing some kind of action on them and *Controller*, which defines the allowed actions that can be performed by processors.

- *Data* represents the information referring to subjects that can be handled by processors. *Data* is extended by means of *Identifiable* data and *Sensitive* data. The former represents the information that can be used to uniquely identify subjects, while the latter represents information that deserves particular care and that should not be freely accessible. Moreover, *Data* is a complex structure composed of basic information units named *Field*. Class *Field* contains the attributes *name* and *content*. The former represents an identifier used to identify the information contained in the field, while the latter describes the information itself.
- *Action* represents any operation performed by *User* (usually *Processor*). Since in a privacy aware scenario a processing is executed under a purpose and an obligation, *Action* is defined as an abstract class that is extended by classes *Obligation*, *Processing* and *Purpose*. Moreover, *Processing* specifies an aggregation relationship with *Purpose* and *Obligation* so that each action can be recursively composed of other actions allowing the definition of complex actions in term of simpler ones. Finally, instances of *Action* are created by instances of *Controller* by means of the services provided by class *FactoryAction*.

Figure 2 provides a complete view of the aforementioned classes along with their relationships. For instance, the dependency relationship between *Action* and *Data* states that data are processed by actions, while the association between *Subject* and *Data* represents data ownership.

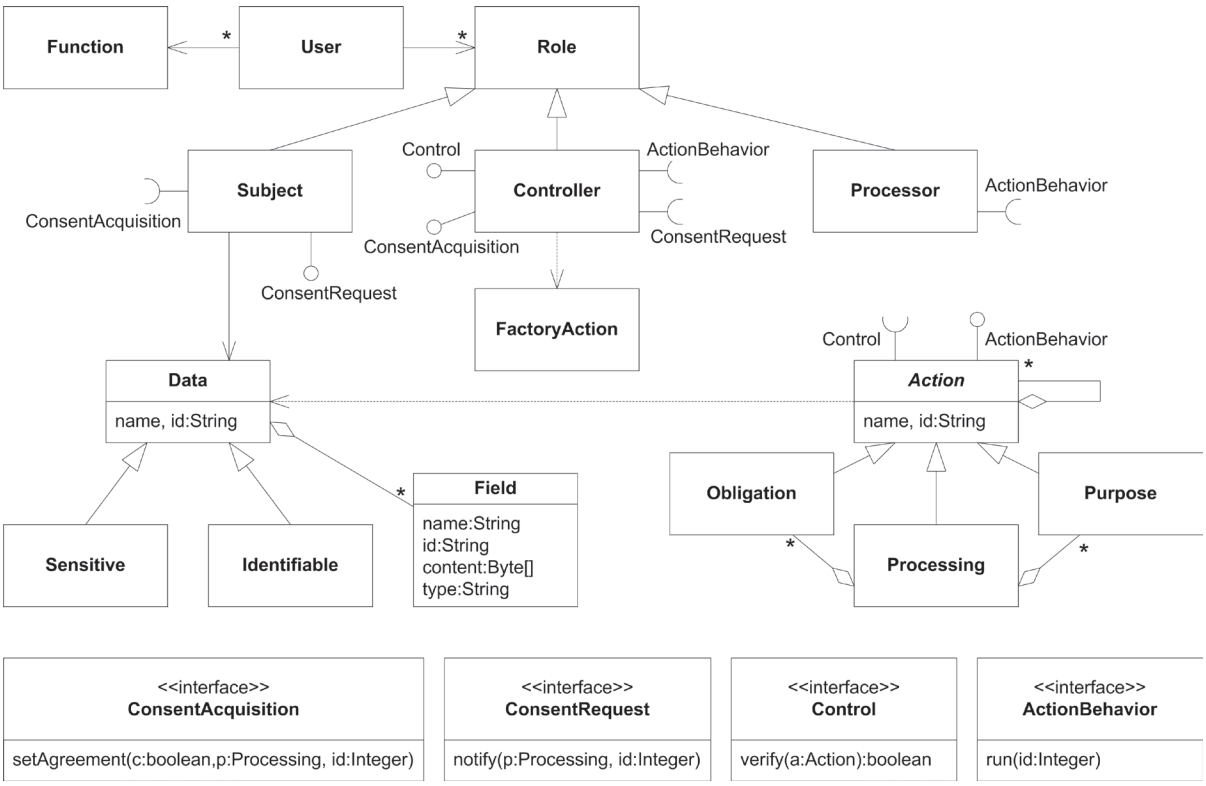
Notice that this model can be extended in order to support the definition of policies related to different application domains. For example, to specify privacy policies compliant with the Italian privacy legislation[6], it is necessary to extend the model introducing the concept of "judicial data". Such an extension can be easily obtained by introducing a class *Judicial* that extends class *Data*.

Furthermore, several interfaces have been introduced to model the flow of information among the instances of the different classes. In fact, an interface defines the services that a class can either implement or use (invoke).

The interface *ActionBehavior*, provided by class *Action*, is introduced in order to model action execution. *ActionBehavior* can be used by classes *Processor* and *Controller* that can invoke the method *run()* to represent the execution of an action. Notice that, each class extending *Action* inherits interface *ActionBehavior* and therefore may provide a specific implementation of method *run()*.

The interface *ConsentRequest*, provided by class *Subject*, is used to notify subjects of both the purposes and the obligations of any processing of their data. Thus, an instance of class *Controller*, taken its *id* and an instance of *Processing*, invokes the method *notify()* of interface *ConsentRequest* to notify a given instance of *Subject* that a processing on his/her data may occur. Interface *ConsentAcquisition*, provided by class *Controller*, is used by class *Subject* to allow its instances to grant or deny the consent to data processing. More specifically, the interface provides the method *setAgreement()*, that taken an instance of *Processing*, the *id* of *Subject* and a boolean value, notifies the controller whether the subject has granted or denied the consent to data processing.

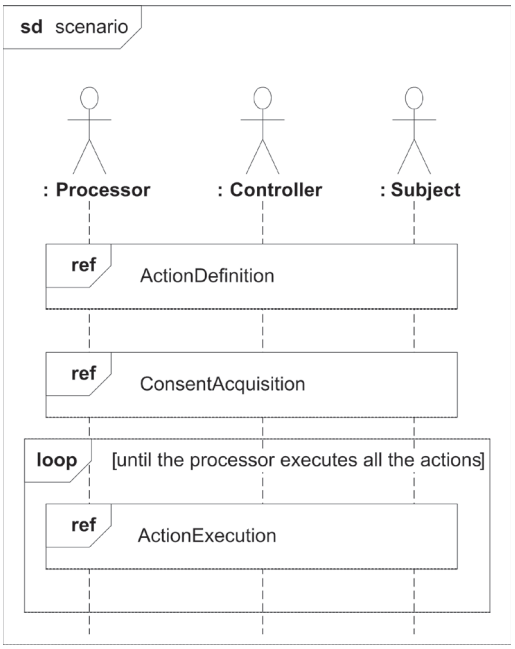
Figure 2. The class diagram that describes the conceptual model



Finally, interface *Control*, defined by class *Controller*, is used by class *Action* to verify whether a given action can be executed, that is whether the subjects involved have granted the consent. Thus, interface *Control* provides the method *verify()* that, taken an instance of *Action*, returns whether the latter is authorized that is, the consent has been granted.

Notice that this model is meant to describe all the activities related to privacy even though some of them may occur outside the system. For example let us suppose that a new customer wants to open a checking account in a traditional bank. In this case he/she may interact with a bank employee who will provide the customer with all the information concerning the privacy policy. Before actually registering the new customer in the system, the employee asks him/her to sign a statement in which the customer grants the consent

Figure 3. The general scenario



to data processing. Such an interaction is not supported by the system since it takes place by oral explanations and documents reading. However, from our point of view this corresponds to having the employee invoking the method *notify()* of *ConsentRequest* and then the customer invoking the method *setAgreement()*.

On the contrary, if we consider a new customer of an on-line bank the system will support all the interactions needed to get the informed consent. In both cases the way in which the interaction is modeled does not change, the difference being to which extent the activities are directly supported by the system rather than being specified by human executed procedures.

Although the class diagram of Figure 2 faithfully represents the components of privacy policies, it does not express any dynamic aspect. This can be done by means of UML Sequence diagrams and therefore in what follows we present some Sequence diagrams modeling general interaction schemes that are common to any privacy aware system. Notice that such diagrams can be specialized and extended for specific needs.

The sequence diagram of Figure 3 reports a general scenario that introduces the main actors along with the basic activities that can occur in a privacy aware system. The scenario refers to three different tasks, named *ActionDefinition*, *ConsentAcquisition* and *ActionExecution*, each of which is represented by means of a Sequence diagram. According to the semantics of UML Sequence diagram, such internal scenarios are sequentially executed.

The first task, named *ActionDefinition*, describes how it is possible to define new actions. Notice that actions can be exclusively defined by users characterized by the role of *Controller*, using the services provided by class *FactoryAction*.

Class *FactoryAction* provides several methods to allow controllers to create new actions. In particular the class allows the definition of the following basic actions:

- Data creation. The method *defData(Data obj, String fieldName, String fieldType, String fieldId)* returns a new basic action whose execution creates an instance of class *Field* associated with *obj*. For instance *defData(d1, "family name", "String", "001"),* inserts a new data field named "family name" of type String into d1 (an instance of class *Data*).
- Data writing. The method *defWrite(Field f, Byte[] content)* returns an action whose execution updates the content of field *f*. For instance, the method *defWrite()* allows one to initialize the content of the previously created field, representing the family name, to the value "Smith".
- Data reading. The method *defRead(Field f)* returns an action whose execution reads the content of the field *f*.

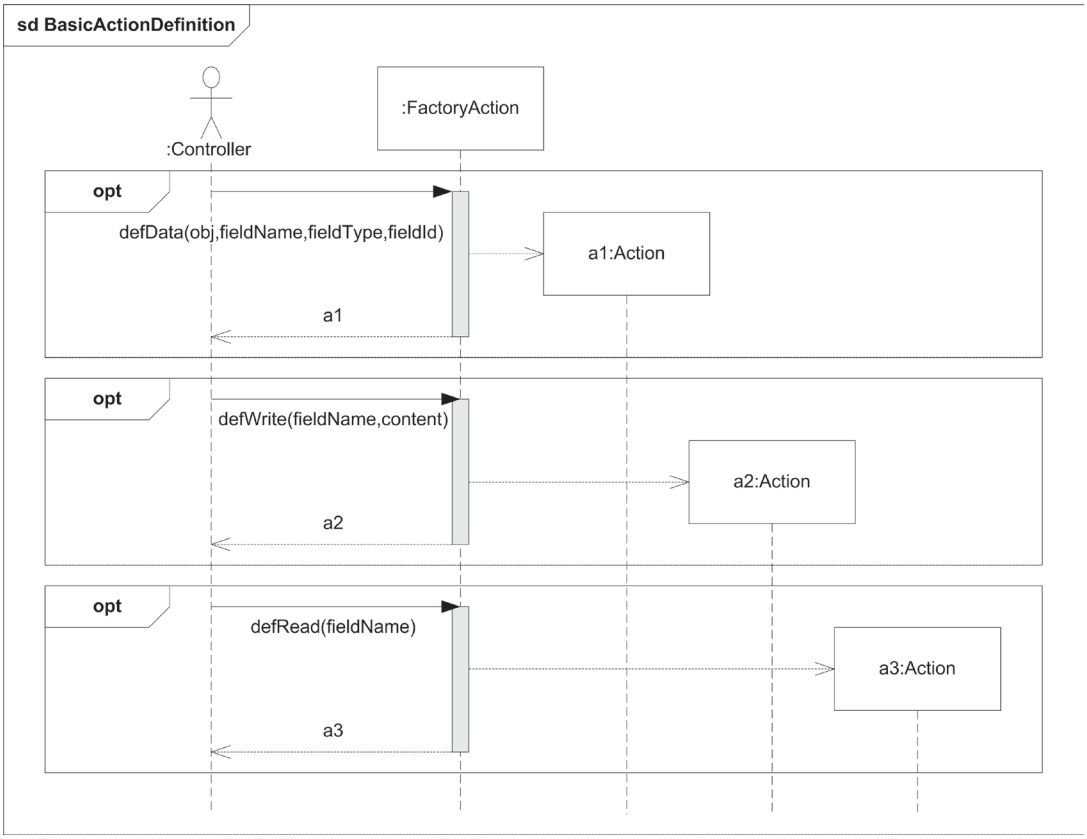
The definition of basic actions is described in the Sequence diagram, named *BasicActionDefinition*, shown in Figure 4.

Such basic actions represent the behavioral unit for the definition of *Processing*, *Obligation* and *Purpose*. Once all the required basic actions are defined, the controller defines a new complex action by invoking the method *defAction(...)*, provided by *FactoryAction (seeFigure 5)*. The method *defAction(...)* takes as input the list of actions composing the new action along with its type (i.e., *Purpose*, *Obligation* or *Processing*). Notice that a *Processing* action is defined by composing instances of *Purpose* and *Obligation*, hence the purpose and the obligation associated with any instance of *Processing* can be easily retrieved.

Actions definition can be carried out by means of two different scenarios. The first one, shown in Figure 6, represents the most common scenario in which, all the needed actions are predefined by a controller.

In fact, for each application domain it is possible to identify a set of actions that almost every

Figure 4. The basic actions definition scenario

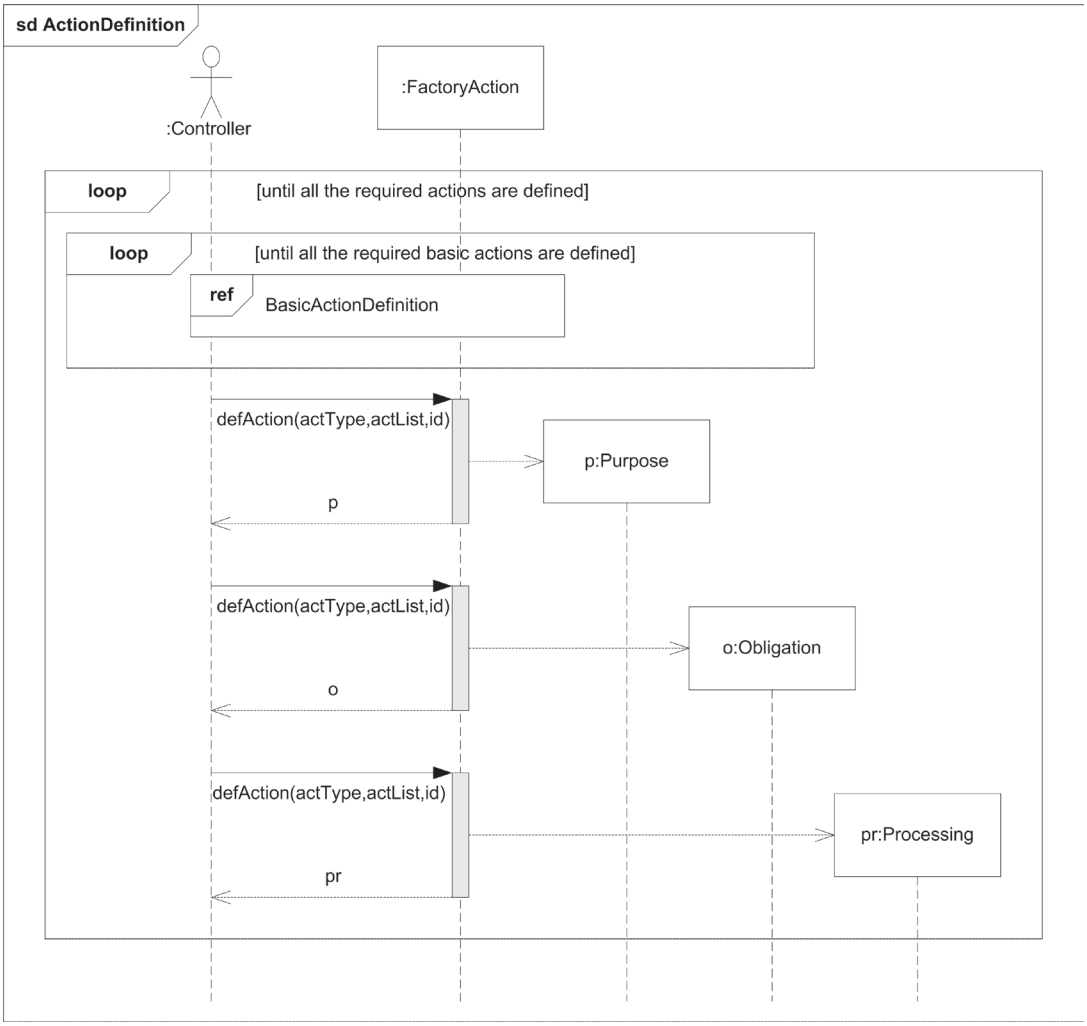


processor will try to execute in order to carry out his/her duty. Such actions represent the services that the company provides to its customers (subjects). In this case the subject consent is acquired *a priori*. As an example, let us consider the case of a potential bank customer that wants to open a checking account. The customer is informed that if he/she will request to make a domestic bank transfer, his/her data will be processed for the purpose of complying with his/her request under the obligation of notifying national authorities whenever the transferred amount exceeds a given threshold. Notice that in this scenario the customer is informed and required to grant consent even if he/she will never request any bank transfer to be made.

In the second scenario, shown in Figure 7, the action definition is triggered by a processor that needs to execute a not yet defined action. Therefore, this scenario describes a situation in which specific actions are built in order to fulfill a specific request coming from a processor; in this case the subject is required to grant consent for any action for which the consent was not granted *a priori*. In this case, the processor interacts with the controller in order to define the purpose, the obligation and the needed processing. In particular, the processor sends his/her *Id* to the controller, which, in turn, instantiates three actions (a purpose, an obligation and a processing) requested for processing the data. Notice that *Processing* represents the intention to perform a specific data processing, which can be carried out only after the involved subject has granted the consent.

As an example, let us consider the case of an actual bank customer requesting to make an in-

Figure 5. The action definition scenario



ternational bank transfer. Since international bank transfers are less common than domestic ones, the customer was not informed nor he/she granted the consent when he/she opened the checking account. Therefore, when the customer requests the international bank transfer, he/she is informed that his/her data will be processed for the purpose of complying with his/her request under the obligation of notifying the National Security Agency. Notice that in this scenario the customer is informed and required to grant consent only when he/she requests for the first time to make an international bank transfer.

In both scenarios before action execution the *Subject* has been notified of the processing and, as described in Figure 8, he/she can grant or deny the consent.

Finally, the scenarios terminate with the execution of the authorized actions (if any). As specified by the loop construct, depending on the processor needs, actions once authorized, can be executed multiple times.

Figure 6. Action definition, the most typical scenario

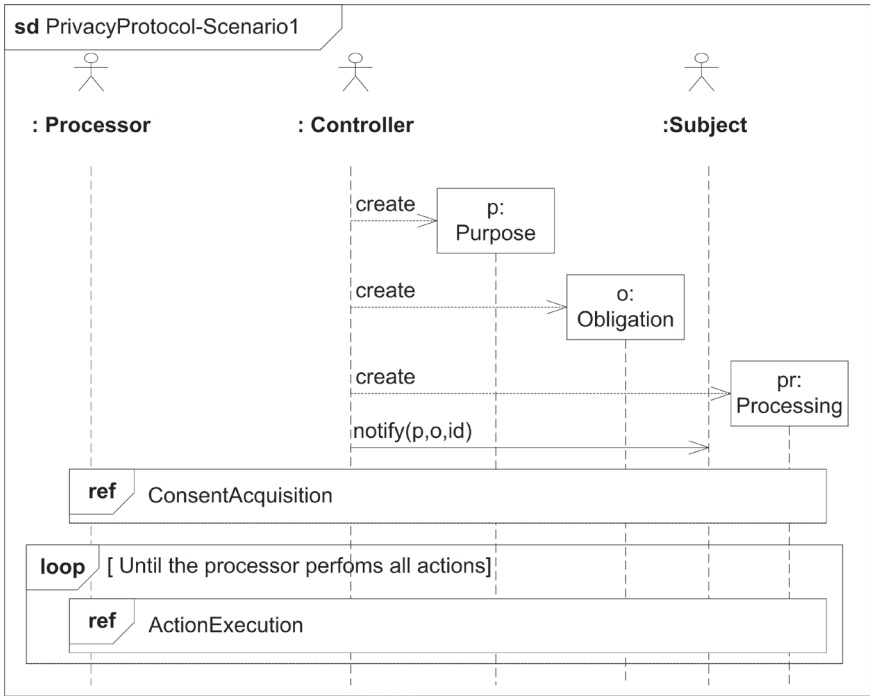


Figure 7. Action definition, the alternative scenario

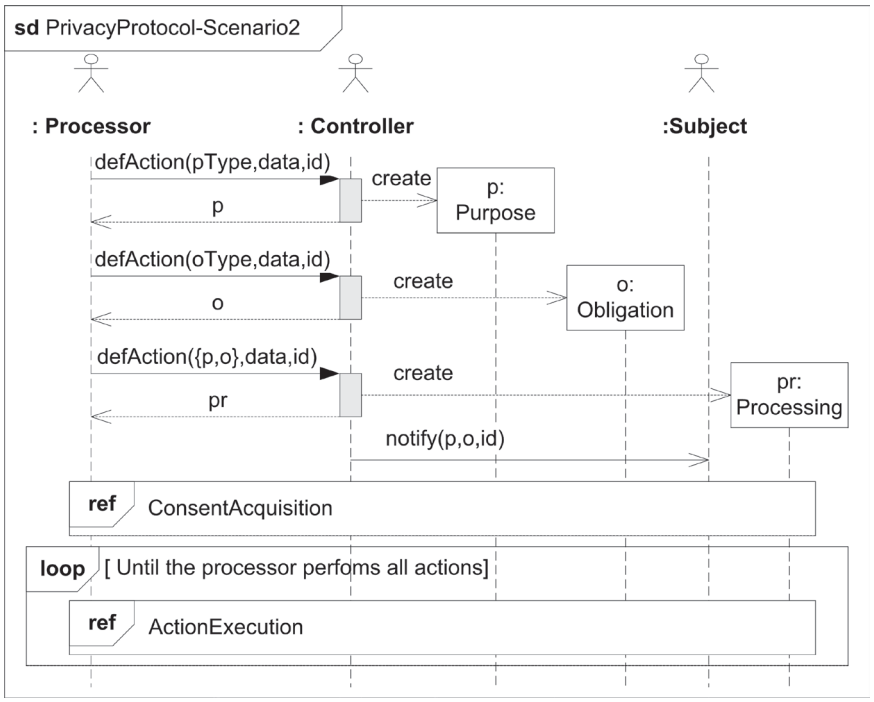
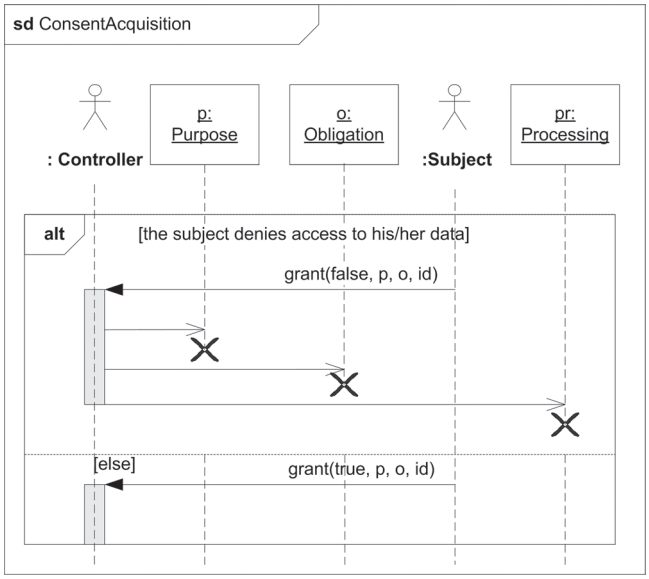


Figure 8. The consent acquisition scenario



TOWARDS DESIGN SOLUTIONS

Anonymity, informed consent acquisition and enforcement of privacy policies are fundamental requirements for privacy aware systems that can be used both to test the effectiveness of the model and to introduce design solutions based on extensions/refinements of the model itself. In what follows we present three different design patterns providing a design solution for each of the above mentioned requirements. The same approach is used to provide solutions to other privacy requirements such as pseudonymity, unobservability and so on. Therefore, privacy requirements can be satisfied by providing appropriate design patterns providing the needed extensions to the conceptual model. Notice that such extensions should be viewed as being part of a development process in which one starts from a high level description and moves towards the solution by adding details concerning the different aspects of a privacy aware system.

Anonymity

Anonymity states that sensitive data managed by a system, cannot be used to retrieve the identity of the data owner, that is the subject to whom data refer, without an explicit authorization. For example, let us consider a hospital information system that stores both health related data (sensitive data) and personal data of hospital patients. Hospital doctors may be allowed to access both kinds of data when they have to make a diagnosis, while if the hospital staff is conducting some statistics, the may be allowed to access only sensitive data without being able to retrieve the identity of patients.

The aim of this pattern is to introduce a domain independent solution schema that drives the construction of anonymity assurance mechanisms, which prevent the identification of individuals starting from their data.

Requirements

Several requirements must be taken into account when defining anonymity assurance mechanisms (Hafiz, 2006):

- Identity masking. Anonymity enabling mechanisms shall mask the identity of subjects.

- Usability. An anonymous data set shall be usable. Extreme solutions such as not releasing any data cannot be accepted. Moreover, anonymity enabling mechanisms shall not alter the processing actions performed by a system.
- Performance. Anonymity enabling mechanisms shall minimally alter the overall system performances.

Solution

The proposed solution starts from the classification of data and users proposed in the conceptual model.

Data Structure

In order to define anonymity, the data handled by a system need to be suitably structured. Data are composed of fields that, depending on their characteristics, are grouped into sensitive and identifiable subsets. Moreover, a data type may be characterized by a hierarchical structure composed of other data types possibly classified as sensitive or identifiable.

In order to keep the link between identifiable and sensitive data, we introduce a reference field to the data structure used for identifiable data. This is done by means class *RefField*, which extends class *Field* of the conceptual model, as shown in Figure 9. The (inherited) attributes of Class *RefField* are used in the following way: the attribute *name* is set to the name of the data to which it refers, while the attribute *content* is set to the value of the attribute *id* of the instance of *Data* to which it refers.

For example, let us consider the definition of a data structure composed of the fields "familyName", "city" and "disease". Fields "familyName" and "city" identify the data owner, while "disease" represents a sensitive information. As a consequence two different data types are defined. The former, named "Person", is composed of the identifiable fields, while the latter, named "Health", contains the sensitive one. Let us consider the following data sets: 1) "Smith", "Milan", "hemicranias"; 2) "Brown", "New York", "gastric ulcer". Therefore, the first triplet is represented by an instance of class *Identifiable* in which the attribute *name* is set to "Person", and the attribute *id* is set to "data001", and by an instance of class *Sensitive* in which the attribute *name* is set to "Health" and attribute *id* is set to"data003". Moreover, "data001" contains an instance of class *Field* characterized by the attribute *name* initialized to "familyName", the attribute *id* initialized to "field001", and the attribute *content* set to "Smith". It also contains a further *Field* characterized by the attribute *name* set to "city", the attribute *id* initialized to "field002", and the attribute *content* set to "Milan". Finally, "data003" contains an instance of *Field* characterized by the attribute *name* initialized to "Disease", the attribute *id* initialized to "field005", and the attribute *content* set to "hemicranias". In order to represent the link between the identifiable data represented by "data001" and the sensitive data represented by "data003", "data001" contains an instance of *RefField* in which the attribute *name* is set to "Health", the attribute *id* is set to "ref001" and the attribute *content* is set to "data003". Figure 10 reports the structure of such data sets by means of a Composite Structure Diagram.

In order to prevent the identification of data owners starting from their sensitive data, instances of *Identifiable* may own references to instances of *Identifiable* or *Sensitive*, while instances of *Sensitive* can own only references to instances of *Sensitive*. In other words, instances of *Sensitive* cannot own any reference to instances of *Identifiable*.

A second issue that must be taken into account concerns the possibility that starting from identifiable data one can access the associated sensitive data, by following the reference fields. However, the system should prevent non authorized users of the system to follow such references, that is there may be some users that can access identifiable data without being authorized to access sensitive data.

Figure 9. Extensions of the conceptual model to support anonymity

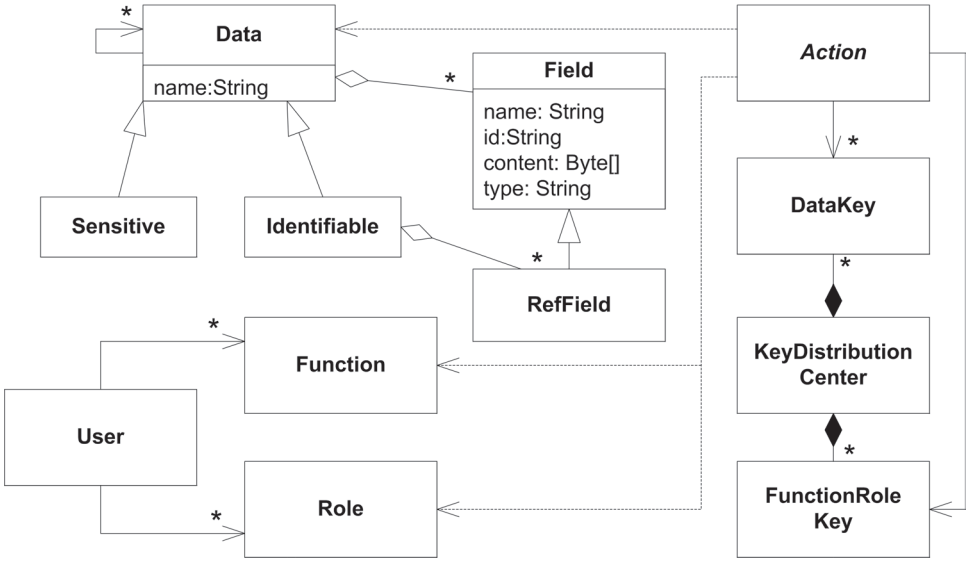


Figure 10. The composite structure diagram that describes the example

data001:Identifiable			data003:Sensible
name = “Person” id= “data001”			name = “Health” id= “data003”
field001:Field	**field002:Field**	**ref001:RefField**	**field005:Field**
name=”familyName” id=”field001” content=”Smith” type=”String”	name=”city” id=”field002” content=”Milan” type=”String”	name=”Health” id=”ref001” content=”data003” type=”Sensitive”	name=”disease” id=”field005” content=”hemicranias” type=”String”

data002:Identifiable			data004:Sensible
name = “Person” id= “data002”			name = “Health” id= “data004”
field003:Field	**field004:Field**	**ref002:RefField**	**field006:Field**
name=”familyName” id=”field003” content=”Brown” type=”String”	name=”city” id=”field004” content=”New York” type=”String”	name=”Health” id=”ref002” content=”data004” type=”Sensitive”	name=”disease” id=”field006” content=”gastric ulcer” type=”String”

The way in which we prevent non authorized accesses to sensitive data is based on cryptography. Notice that at this level we do not need to choose any particular encryption technique (e.g., public key, symmetric key, etc.), since the needed extensions of the conceptual model are independent from encryption techniques.

Handling Cryptography

The way in which we introduced cryptography is based on three new classes (see Figure 9): *KeyDistributionCenter*, *DataKey* and *FunctionRoleKey*. The class *KeyDistributionCenter* manages the generation of the keys usable for encryption purposes. *KeyDistributionCenter* generates keys according to the restrictions imposed by the privacy policy. *FunctionRoleKey* represents the key associated with a specific pair *Function-Role*, while *DataKey* represents the key to encrypt the content of data fields.

In what follows we present the use of the previously introduced concepts for the definition of anonymity mechanisms.

Data Encryption

A key, named *DataKey*, is generated to encrypt the value of the attribute *content* of the reference fields that refer to instances of sensitive data. As an example, let us consider that for statistics purposes we need to know how many people living in Milan suffer from hemicranias. As described above, such data types are separately defined and a reference field, named "Health", is defined on "Person". Notice that the attribute *content* of "Health" is encrypted, and therefore it is not possible to access the sensitive data without knowing the key that is required to decrypt such a field.

Actions

Data can be accessed only by means of actions (see Figure 2). Actions are expressly built to be executed by users that belong to a given function-role pair. In order to guarantee that actions, once defined, can be executed only by authorized users, an authentication mechanism is introduced. More specifically, a key, represented by the class *FunctionRoleKey*, is generated and released to the authorized users.

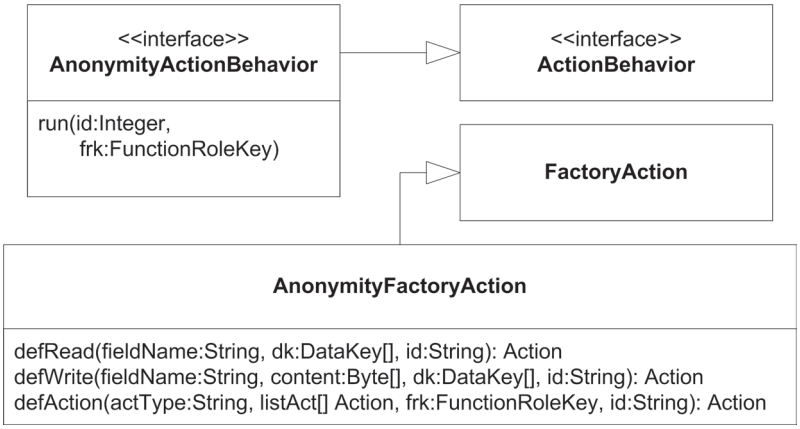
FunctionRoleKey instances are handled by *KeyDistributionCenter*, which provides generation and secure communication mechanisms like the ones proposed by Kerberos[7]. Whenever a user-controller defines a new *Action*, two keys are generated. The former key is associated with the pair *Function-Processor* that is authorized to execute the action, while the latter with the pair *Function-Controller* that has to supervise the execution. Notice that the specification of the algorithm used for key generation, and of the communication protocol is out of the scope of this pattern.

In order to support encryption a new class and a new interface are introduced (see Figure 11). The class, named *AnonymityFactoryAction*, extends class *FactoryAction*, while the interface, named *AnonymityActionBehavior*, extends the interface *ActionBehavior*. *AnonymityFactoryAction* redefines most of the methods inherited from *FactoryAction* (i.e., *defRead()*, *defWrite()* and *defAction()*) by adding a new parameter representing an instance of the encryption/decryption *DataKey* for *defRead()*/*defWrite()* and representing an instance of *FunctionRoleKey* that identifies the authorized users.

For example, let us suppose that a researcher who works in a health care institute wants to know how many people living in Milan suffer from hemicranias. Moreover, suppose that data are organized by means of the structure described in Figure 10. Therefore it is necessary to access the fields "city" and "disease". The controller, in order to create such an action, invokes the method *defAction()* passing as parameter an instance of class *FunctionRoleKey* associated with the pair Researcher/Processor that is authorized to execute the action once defined.

Actions can be executed by invoking the method *run()* (defined by *AnonymityActionBehavior*), providing the key *FunctionRoleKey*, and the *id* of *User*. Notice that users authentication can be carried out in different ways. For example, the first task of method *run()* may check whether the function-role key provided by the user is the same key that was set at action definition time.

Figure 11. Extensions of the conceptual model to support anonymity



Consequences

The pattern has the following benefits.

- Privacy. The separation of sensitive data from identifiable data, and the adoption of encryption techniques makes it more difficult to associate sensitive data with the identity of data owners.
- Minimal user involvement. The users are not required to modify their normal activities.

A not properly defined implementation of this pattern may suffer from the following weaknesses.

- Usability. A too high granularity level of encryption mechanisms can undermine the usefulness of data. As an example, in the case of database applications, if all the data entries are encrypted, the resultant dataset may be hardly used even by authorized users.
- Overhead and delay. The application of encryption mechanisms requires adequate computational resources. Hence, the overall system performances may worsen, and delays and/or overheads can be generated. In order to guarantee an adequate level of usability and privacy, it is necessary to balance the usage of encryption techniques.

Informed Consent

Informed consent states that individuals (i.e., data owners) must be informed on the purposes of any processing involving their data. Therefore, the goal of this pattern is to provide a basic schema to deal with the acquisition of the informed consent.

Requirements

Several requirements need to be taken into account in order to define informed consent acquisition mechanisms:

- Disclosure. The data owner has to be informed of the processing purposes, before processing can take place.
- Agreement. The data owner has to reply to the requests to access his/her data by specifying whether he/she agrees upon.
- Comprehension. The data owner has to state whether he/she understood how the requested information will be used.
- Voluntariness. The data owner has to ensure whether his/her consent is given without any coercion or external influence.

- Competence. The data owner has to declare whether he/she is adequately competent to provide the consent. For example, he/she has to state to be of age.

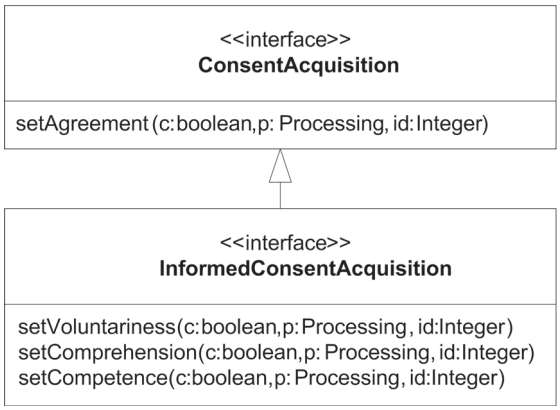
Solution

According to the conceptual model presented in this chapter, data owners are represented by means of class *Subject*, while actions are represented by means of classes *Processing*, *Purpose*, and *Obligation*. Moreover, all the actions involving the acquisition of the informed consent are executed by an instance of class *Controller*. Therefore, the acquisition of the informed consent requires user-subjects and user-controllers to communicate among them using the method of the interface *ConsentAcquisition* provided by class *Controller*. However, in order to deal with the requirements of competence, voluntariness and comprehension it is necessary to extend the interface *ConsentAcquisition* introducing a new interface, named *InformedConsentAcquisition* (see Figure 12).

Such an extension satisfies all the previously introduced requirements, as discussed in what follows:

- Disclosure. In order to inform *User-Subject* of the processing purpose, *User-Controller* invokes the method *notify()* of the interface *ConsentRequest* by specifying the purpose of the processing and under which obligation the action will be executed.
- Agreement. In order to reply to the request of *User-Controller*, *User-Subject* invokes the method *setAgreement()* by specifying whether he/she granted the consent for processing his/her data.
- Comprehension. The *User-Subject*, in order to confirm whether he/she understood how the requested information will be used, invokes the method *setComprehension()*.
- Voluntariness. The *User-Subject*, in order to ensure whether his/her consent is given without any coercion or external influence, invokes the method *setVoluntariness()*.
- Competence. The *User-Subject*, in order to declare whether he/she is adequately competent to provide the consent, invokes the method *setCompetence()*.

Figure 12. The extensions required to support the acquisition of the informed consent



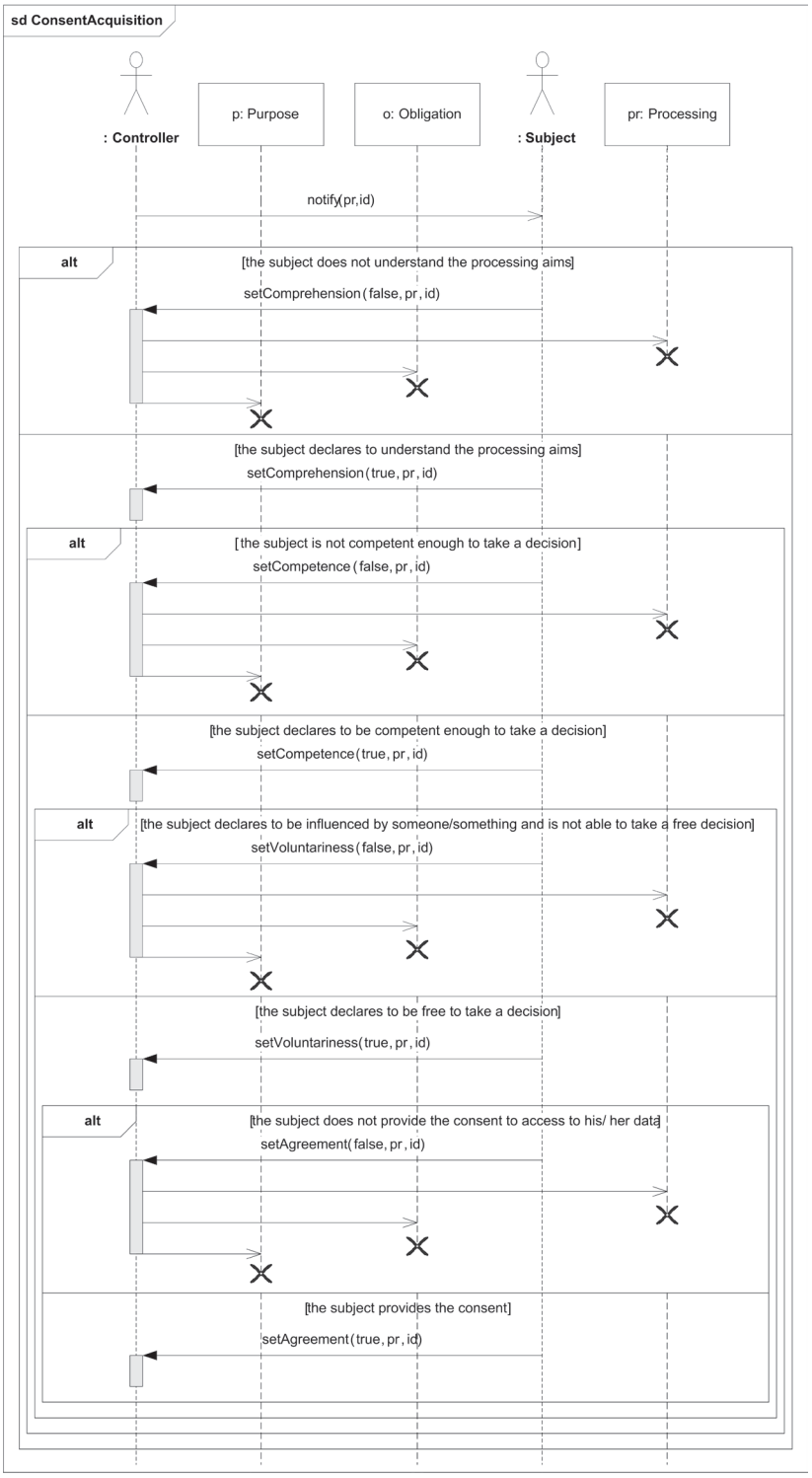
The UML Sequence diagram of Figure 13 describes a consent acquisition scenario. Notice that the sequence of actions proposed is only one of the many scenarios that can be defined. In other words, *Subject* has to invoke all the methods of the interface *InformedConsentAcquisition*. In case *Subject* does not grant his/her consent, all the actions that were defined for accessing his/her data are destroyed.

Consequences

This pattern offers the following benefits:

- Trust: the exchange of clear and complete information increases the individuals confidence in the system.
- Protocol: the proposed pattern, besides defining the fundamental interactions among the actors involved in a consent acquisition scenario, supports the definition of different interaction protocols.

Figure 13. The consent acquisition scenario



This pattern may suffer from the following weaknesses:

- The pattern can not assure that the system will comply with the obligations under which the consent is given. Notice that this is a requirement for the pattern Enforcement.
- The declarations of comprehension, voluntariness and competence depend on the user. Since human behavior is unpredictable, the declarations may not reflect the truth and they cannot be directly verified.
- The exchange of messages may worsen the overall system performance: delays and/or overheads may be generated.

Enforcement

Privacy aware systems prevent the unregulated disclosure of data by means of access control mechanisms. Although such mechanisms regulate data access they cannot assure that the processing activities comply with the stated purposes, nor with the stated obligations that were given to a *Subject* at consent acquisition time.

This privacy pattern tries to address such issues, focusing on the definition of enforcement mechanisms that aim at verifying the compliance of the processing activities with the privacy policy.

Requirements

Once a data owner granted the explicit consent the system has to guarantee that processing is compliant with the stated purposes. Thus, it is necessary to provide a way to verify processing compliance. In principle there are two different ways in which such a verification can be carried out: run-time verification and *ex-post* verification. Run-time verification requires that every action is checked before actual execution, while *ex-post* verification requires that actions are verified once they are executed (e.g., audit-based mechanisms). Thus, the former aims at preventing the execution of actions that are not authorized, while the latter aims at analyzing the system evolution in order to find any possible unauthorized processing.

Solution

All the actions required by a privacy policy are defined as instances of classes *Purpose*, *Obligation* and *Processing*, which are extensions of the abstract class *Action*. Class *Action* uses interface *Control* that, in turn, defines the method *verify()* to carry out the verification of the compliance of any instance of *Action* with a given policy.

Actions are executed by *Processor* by invoking the method *run()* of the abstract class *Action*, while verification is carried out by *Controller*.

In what follows we discuss both run-time and *ex-post* verification.

Run-Time Scenario

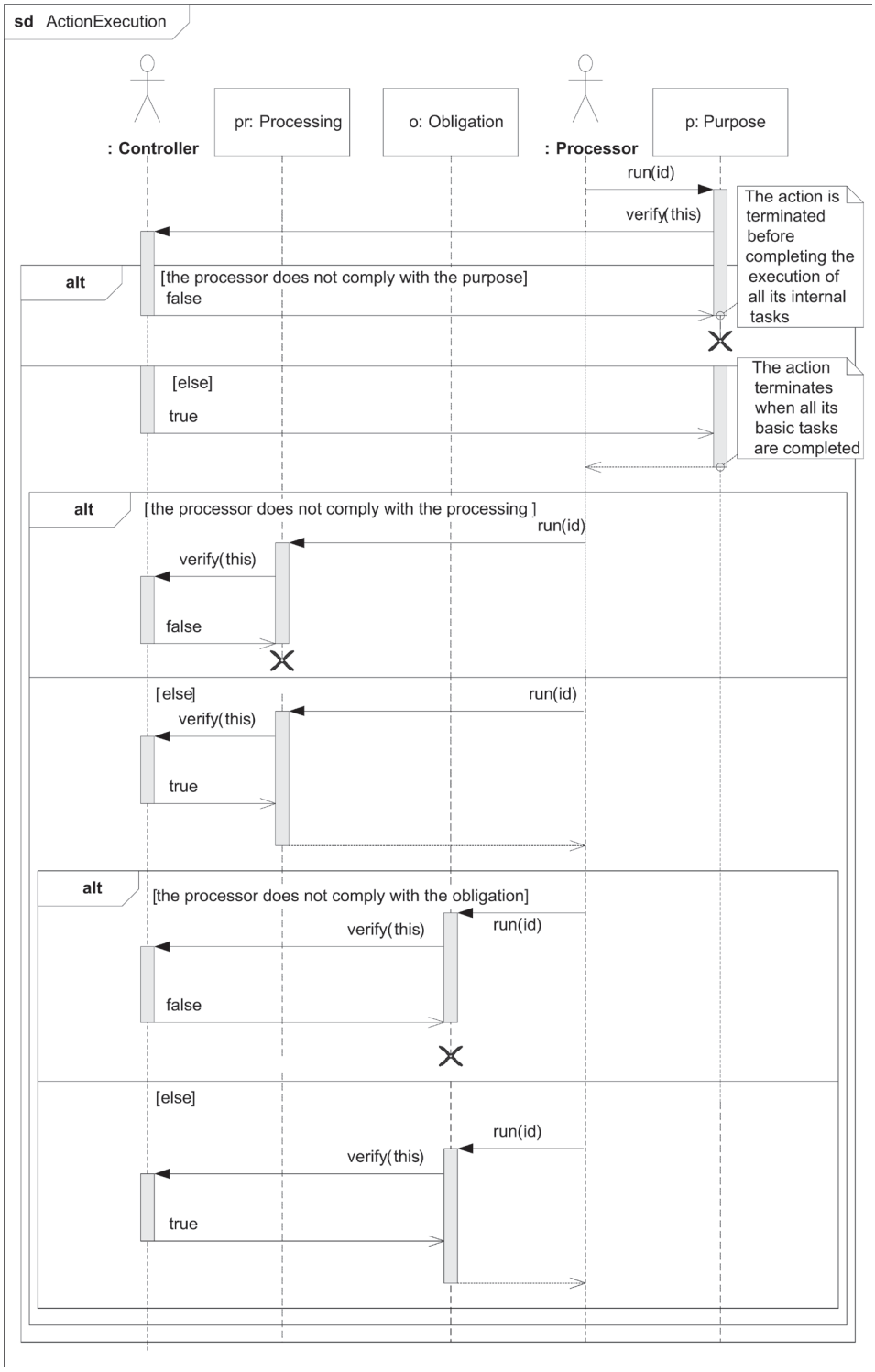
A *Processor*, in order to execute an action invokes the method *run()* that, in turn, invokes the method *verify()* thus allowing *Controller* to check whether the action is compliant with the privacy policy. If not, *Controller* prevents *Processor* from executing any further action, as described in the *ActionExecution* scenario reported in Figure 14. Notice that the enforcement mechanism cannot oblige *Processor* to perform the required obligations, if any.

Ex-Post Scenario

In an *ex-post* enforcement scenario, the verification of the compliance of the actions executed by *Processor* is performed after their actual execution.

Verification is carried out as in the run-time scenario, that is *Controller* invokes the method *verify()*. However, in this case *Controller* cannot prevent *Processor* from executing unauthorized actions, but the non compliance can be recorded so that *Processor* can be prevented from executing other actions.

Figure 14. Enabling run-time enforcement



Consequences

This pattern offers the following benefits:

- Generality. This solution is general enough to be applied to different application domains.
- Performance. *Ex-post* verification does not affect the overall system performances, while run-time verification may worsen the performances of the actions that need to be verified.
- Preserving privacy: Run-time verification prevents privacy violations to occur, while *ex-post* verification may result in privacy violations that will not be discovered until verification takes place.

This pattern suffers from the following weaknesses:

- Independence. This pattern does not address the definition of the activities performed by the *verify()* method. Such activities strictly depend on the characteristics of the system and of the actions to be verified.
- Overhead and delay. Run-time verification requires adequate computational resources. As a consequence, the overall system performances may worsen.

AN EXAMPLE

In order to assess the model presented in this chapter, we discuss an example of its application in the field of healthcare.

Hospital Information Systems are a fundamental tool for healthcare organizations since they support the management of the most important and characterizing internal processes of a hospital structure. Such systems provide different types of services such as: patient registrations, physical examination reservations, patients' admission, etc. All these services handle sensitive and personal data of the patients and of the personnel that operate in the hospital structure, hence a particular care to the management of such data is required.

The rest of this section provides a simple example concerning the definition of a privacy policy for data management in a Hospital Information System of a diagnostic centre.

A diagnostic centre is an organization where different actors operate. The following functions are considered in our scenario: doctor, nurse, employee, laboratory technician and outpatient.

- Outpatients need to be medically assisted. They request a physical examination with a medical specialist or a diagnostic test, and once examined they pay the fee
- Doctors examine outpatients, access and modify their case histories, prescribe therapies or other medical examinations.
- Nurses execute specific actions such as taking a sample of blood, or specific physical examination such as measuring the blood pressure.
- Employees perform bureaucratic activities: such as registering outpatients, making appointments for medical examinations, preparing purchase orders.
- Laboratory technicians are specialized personnel that execute diagnostic tests and draw up the medical report.

The Privacy Policy

Data processing has to be regulated by policies that specify 1) who is allowed to process data, and 2) what can be done with such data. The system manages different types of data:

- Patient case histories, which are detailed records on the background of a person under treatment.

- Medical examination prescriptions: the requests of thorough diagnostic tests.
- Medical examination results: the results of the diagnostic tests.
- Identifiable data: identifiable data associated with patients
- Administrative data: the payment state for medical examinations and treatments.

In what follows we consider one of the activities that are usually supported by a HIS, namely blood tests management.

Let us consider that an outpatient needs to contact the diagnostic center to request an appointment for a blood test.

The following scenario sketches the involved actors and actions:

- An employee makes the appointment for the medical examination;
- The outpatient goes to the appointment, and a nurse takes a sample of his/her blood;
- The outpatient pays the fee at the payment office;
- A laboratory technician executes the blood test and stores the results of the patient in the system;
- The patient picks up the results.

The privacy policy that we wants to model must satisfy the following requirements:

- Outpatients must be informed of the processing purposes of the diagnostic centre.
- The processing of the data of the outpatients is exclusively allowed under their explicit consent.
- The system has to prevent the identification of outpatients starting from their health related data.
- The processing actions can be exclusively executed by authorized users

Modeling the Example

The actors involved in the proposed scenario are represented by means of instances of the classes *User*, *Function* and *Role*. Notice that in a real scenario users may be characterized by multiple function-role associations, but in this example we do not consider this situation for the sake of simplicity.

Employees are instances of *User* characterized by a *Function* that specify the task of "Employee" and by the role *Processor*, since employees process the data of the patients. Similarly, doctors are characterized by the *Function* "Doctor" and the role of *Processor*, while technicians are characterized by the *Function* "Laboratory Technician" and the role of *Processor*. Finally, outpatients are the data owners whose data will be processed by doctors, laboratory technicians and employees. Therefore, outpatients are characterized by the role of *Subject* and no *Function* is associated with them.

Data

The data managed by the system concern outpatients, appointments, costs, payments and the results of diagnostic tests. Moreover, the system should keep track of which laboratory technician executes a diagnostic test for a given outpatient.

First of all it is necessary to model the data structure and to classify the different *Data* instances as sensitive or identifiable. In particular the following data types (i.e. instances of class *Data*) are introduced:

- "Person": composed of fields such as "first name", "family name", "birth date", "address", "telephone number", "social security number", which identify an outpatient.
- "Physical examination", "Diagnostic test": composed of fields such as: "date", "time", "place", "examination type", "examination description", which provide information on the examination/test.

- "Price list": composed of fields such as "examination type" and "price", which describe the price associated with each examination.
- "Result": composed of fields that describe the results of the examination / test.
- "Processor trace": composed of fields that keep track of the users that performed the examination / test.
- "Payment information": composed of fields that keep track of the payment of the examinations/tests.

"Person" is an identifiable data type that stores references to instances of sensitive data such as "Diagnostic test" / "Physical examination". Moreover, "Physical examination"/ "Diagnostic test" stores a reference to further sensitive data named "Result", "Processor trace" and "Payment information". "Price list" is neither sensitive not identifiable data type.

Actions

In what follows we introduce the actions needed to model blood tests management.

- "Registration": registers a new outpatient into the information system of the diagnostic centre
- "Physical examination reservation" / "Diagnostic test reservation": makes a reservation for an examination / diagnostic test
- "Log Processor": keeps track of the processor that executes specific actions
- "Record Result": stores the tests result of the outpatient
- "Pay the bill": stores the payment of the fee associated with a test or with an examination
- "Check payment": verifies that a fee was paid
- "Print result": prints out the results of an examination

In order to assure anonymity we assume that actions are executed only by authorized users. Moreover, we assume that a key distribution centre and a key management service exist so that encryption keys can be created and distributed to each pair function-role that operates the system. In particular, the keys are used to encrypt/decrypt the reference fields needed to access sensitive data. The choice of a specific encryption/decryption algorithm is not discussed being out of the scope of this chapter.

For each action it is necessary to specify: 1) the data types and the fields that need to be accessed; 2) the keys that are needed to access reference fields; and 3) which pair *Function-Role* can execute the action along with the needed keys.

Scenarios

In the following we introduce several scenarios, each of which describes the way in which the interactions between the different actors and the system occur. In particular, the scenarios taken into account concern:

- The acquisition of the informed consent from the outpatient.
- The registration of the outpatient.
- The way in which appointment for blood tests are made.
- The way in which the blood sample is taken from the outpatient.
- How the outpatient pays the fee.
- The activities related to the blood test examination.

Acquiring the Consent

Before any processing concerning the outpatient data can take place, he/she has to grant the informed consent. Therefore an explanation of the process-

ing purposes must be provided to outpatients. Notice that this scenario follows the Informed Consent Pattern.

Whenever a new outpatient enters the diagnostic centre, an employee at the registration desk asks the patient to provide the consent to process his/her data. Thus the employee acts as *Processor*, while the outpatient acts as *Subject*.

The employee informs the patient by means of the method *notify()* provided by interface *ConsentRequest*. Once done, the outpatient is informed of the actions (processing, purpose and obligations) that the system of the diagnostic center may execute on his/her data.

Then, the outpatient interacts with the information system by specifying his/her comprehension, competence, voluntariness and agreement by means of the methods *setComprehension()*, *setCompetence()*, *setVoluntariness()* and *setAgreement()* provided by interface *InformedConsentAcquisition*. Since no action can be executed on the data of an outpatient if he/she does not grant the consent, the following scenarios can take place only if the consent was granted.

Registering the Outpatient

The employee records the data of the outpatient by means of the method *run()* of the action "Registration". In order to execute the method *run()* the employee has to specify his/her *id* and his/her *FunctionRoleKey*. In order to verify that the employee is authorized to execute the action, the method *run()* checks whether the provided *FunctionRoleKey* equals the one introduced at action definition time. If the key is the same the action is executed and the data of the outpatient are stored in the system, that is a new instance of "Person" is created, its attribute *id* is initialized and the value is communicated to the outpatient. Otherwise the execution is aborted since the employee is not authorized to execute the action.

Notice that this check can be considered as a run-time enforcement in which the role of the controller is played by the system rather than by a physical person.

Making the Appointment

Once the outpatient is registered, the employee can make an appointment for the blood test by executing the action "Diagnostic test reservation". This action is composed of multiple basic actions (data writing/data reading) involving some of the fields of "Person", "Price list" and "Diagnostic test". More specifically, the action requires to access the fields "first name", "family name", "birth date" and "social security number" of "Person" and the fields "examination type" and "price" of "Price list".

The action defines a new instance of "Diagnostic test" and initializes the fields "date", "time", "place", "examination type" and "examination description". The action requires the employee to provide his/her function-role-key, so that the keys required to access the reference fields of the involved data sets ("Diagnostic test" of "Person", and "Payment information", "Result" and "Processor trace" of "Diagnostic test") can be automatically retrieved from the key distribution centre. Therefore the action is executed by means of the method *run()* by specifying the identifier of the employee, his/her *FunctionRoleKey*, the identifier of the outpatient, the type and a description of the diagnostic test, the date, the time and the place of the examination. As in the previous scenario, the system checks whether the employee is allowed to execute the action. The action creates a new instance of "Diagnostic test", whose *id*, once encrypted using the key associated with the sensitive data "Diagnostic test", is stored in the homonymous reference of the instance of "Person" representing the outpatient. The action also creates an instance of "Payment informa-

tion", "Result", and "Processor trace". "Payment information" specifies the total amount due for the examination, while "Result" will be used by the laboratory technician to store the results of the examination. Finally, "Processor trace" is used to keep track of the examination executors. Notice that this action initializes only the field *id* of "Result" and "Processor trace", and the fields *id* and *total* of "Payment information" while all the other fields will be set during the execution of other actions.

The values of *id* are encrypted with the keys associated with sensitive data "Payment information", "Result" and "Processor trace", respectively, and the resulting values are stored in the homonymous reference fields of "Diagnostic test".

When the execution completes, the diagnostic test is booked.

Taking a Sample of Blood

The outpatient gives the *id* of the reservation to the nurse in charge of taking the blood sample. Once the blood sample is taken the nurse labels the test tube with the *id* of the test (i.e., the value of the attribute *id* of "Diagnostic test"). Then he/she registers the test by invoking the action "Log Processor", which creates an instance of "Processor trace" and initializes the value of the field "executor" with the *id* of the nurse.

Notice that also in this scenario the system checks whether the action is executed by an authorized member of the staff (i.e., the nurse). As usual this is done by means of the key associated with the pair *Nurse-Processor* that has to be provided when executing the action.

Paying the Fee

The outpatient has to pay the fee for the execution of diagnostic test. Hence, he/she provides the *id* of the reservation to the payment office employee that, in turn, registers the payment by invoking the action "Pay the bill". This action accesses the field "examination type" of "Diagnostic test" and the fields "examination type" and "price" of "Price list". The value of the field "examination type" of "Diagnostic test" is used to calculate the price associated with the examination. The resulting value is used to update the field "paid" of "Payment information".

Examining the Sample

The laboratory technician executes the diagnostic test on the sample of blood. Once done, he/she executes the action "Log Processor" to keep trace of the technician who did the test. The action uses the *id* written on the label of the test tube and the *id* of the technician. Finally, the technician stores the results of the test by executing the action "Record test results", which sets the values of all the fields of the data type "Result".

At this point the outpatient may get the results by providing the *id* of the "Diagnostic test" to the employee. The employee checks whether the outpatient paid the fee by means of the action "Check payment". During action execution the key to decrypt the field "Payment information" of "Diagnostic test" is retrieved. Once decrypted, the value is used to verify whether the outpatient has paid the amount due. If this is the case, the employee invokes the action "Print Results", otherwise he/she notifies the outpatient that he/she still owes some money to the diagnostic centre.

Action "Print Results" decrypts the reference field "Result" of "Diagnostic test" to access the instance of "Result" so that the complete report can be printed and handed to the outpatient.

CONCLUSION

Privacy is becoming more and more important in many aspects of every day life, and therefore it is

becoming a fundamental requirement in the development of systems that handle individuals data.

In this chapter we presented an UML-based conceptual model for the definition of general privacy policies, allowing one to define the concepts needed to deal with privacy-related information. The choice of using UML is motivated by the fact that it is well known by a wide range of analysts, modelers and programmers and therefore the model can be easily understood. Moreover, UML supports a model centric development process and thus the different diagrams introduced in this chapter provide different views showing the main aspects of the whole model. Finally, UML can be used for representing concepts at different levels of abstraction. Even though the presented model has a high level of abstraction, it can be easily extended and adapted for specific application domains;

This chapter also describes some design solutions for specific privacy related recurrent problems. More specifically, the chapter presents a general solution to implement anonymity, to support the informed consent and to define enforcement mechanisms.

The model provides the conceptual foundations that are required by such problems, such as the separation of sensitive from identifiable data, and the classification of roles and actions. The proposed solutions, represented by means of design patterns, consist in concepts and guidelines that drive the modeler towards the definition of privacy aware systems. The solutions extend the conceptual model by adding the elements, such as data encryption, needed to support the above mentioned requirements.

An example concerning the healthcare domain presents the application of the patterns. The example drives the reader through the classification of users and actions, and shows how it is possible to integrate the encryption mechanisms in order to define anonymity and how it is possible to support the informed consent and the enforcement.

REFERENCES

Agrawal, R., Bird, P., Grandison, T., Kiernan, J., Logan, S., & Rjaibi, W. (2005). Extending Relational Database Systems to Automatically Enforce Privacy Policies. *Int. Conf. on Data Engineering (ICDE 2005)*, (pp. 1013-1022). IEEE Computer Society.

Anton, A. (1996). Goal-Based Requirements Analysis. *IEEE Int. Conf. on Requirements Engineering (ICRE 96)*, (pp. 136-144). Colorado Springs CO.

Blakley, B., & Heath, C. (2004). *Security Design Patterns*. Technical Guide, The Open Group.

Chung, E. S., Hong, J. I., Lin, J., Prabaker, M. K., Landay, J. A., & Liu, A. L. (2004). Development and evaluation of emerging design patterns for ubiquitous computing. *Int. Conf. on Designing Interactive Systems*, New York: ACM Press.

Chung, L. (1993). Dealing with Security Requirements during the Development of Information System. *Int. Conference on Advanced Information System Engineering (CAiSE '93)*, Paris (France).

Coen-Porisini, A., Colombo, P., Sicari, S., & Trombetta, A. (2007). A Conceptual Model for Privacy Policies. In Proc. of *Software Engineering Application (SEA '07)*. Cambridge, MS.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA: Addison-Wesley.

Hafiz, M. (2006). A collection of privacy design patterns. *Int. Conf. on Pattern Languages of Programs Conference (PLOP)*. New York, NY, USA.

Kavakli, E., Kalloniatis, C., Loucopoulos, P., & Gritzalis, S. (2008). Addressing Privacy Requirements in System Design: the PriS Method. [New York: Springer]. *Journal Requirements Engineering*, *13*(3), 241–255. doi:10.1007/s00766-008-0067-3

Lamsweerde, A. V., & Letier, Handling, E. (2000). Obstacles in Goal-Oriented Requirement Engineering. *IEEE Transactions on Software Engineering*, *26*, 978–1005. doi:10.1109/32.879820

Liu, L., Yu, E., & Mylopoulos, J. (2002) Analyzing Security Requirements as Relationships among Strategic Actors. *Symposium on Requirements Engineering for Information Security (SREIS '02)*. Raleigh, North Carolina, USA.

Mielikinen, T. (2004). Privacy Problems with Anonymized Transaction Databases. *Int. Conf. on Discovery Science (DS 2004)*, Vol 3245 of Lecture Notes in Computer Science 3245, Springer.

Mouratidis, H., & Giorgini, P. (2007). Secure Tropos: A Security-Oriented Extension of the Tropos methodology. [IJSEKE]. *International Journal of Software Engineering and Knowledge Engineering*, *17*(2), 285–309. doi:10.1142/S0218194007003240

Mouratidis, H., Giorgini, P., & Manson, G. A. (2003b). An Ontology for Modelling Security: The Tropos Approach. *Int Conf. on Knowledge-Based Intelligent Information & Engineering Systems (KES 2003)*, Vol. 2773 of Lecture Notes in Computer Science, (pp. 1387-1394). Springer.

Mouratidis, H., Giorgini, P., & Mason, G. A. (2003a). Integrating Security and Systems Engineering towards the Modelling of Secure Information System. *Int. Conf. on Advanced Information System Engineering (CAiSE '03)*, Vol. 2681 of Lecture Notes in Computer Science, (pp. 63-78). Springer.

Mylopolulos, J., Chung, L., & Nixon, B. (1992). Representing and Using non Functional Requirements: a Process Oriented Approach. *IEEE Transactions on Software Engineering*, *18*, 483–497. doi:10.1109/32.142871

Narayanan, A., & Shmatikov, V. (2005). Obfuscated Databases and Group Privacy. *ACM Int. Conference on Computer and Communications Security (CCS '05)*, (pp. 102-111). New York, NY, USA. ACM Press.

Ni, Q., Trombetta, A., Bertino, E., & Lobo, J. (2007). Privacy-aware Role-Based Access Control. *ACM Symp. on Access Control Methods And Technologies (SACMAT '07)*. Sophia Antipolis, France.

Romanosky, S., Acquisti, A., Hong, J., Cranor, L., & Friedman, B. (2006). Privacy Patterns for Online Interactions. *Int. Conf. on Pattern Languages of Programs Conference (PLOP)*. New York, NY, USA.

Schumacher, M. (2002) Security Patterns AND Security Standards. *European Conf. on Pattern Languages of Programs (EuroPLoP)*, Kloster Irsee, Germany

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., & Sommerlad, P. (2006). *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons.

Schummer, T. (2004). *The Public Privacy–Patterns for Filtering Personal Information in Collaborative Systems, (Tech. Rep)*. Hagen, Germany: FernUnivesität in Hagen.

Steel, C., Nagappan, R., & Lai, R. (2005). *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall.

Yoder, J., & Barcalow, J. (1997). Architectural Patterns for Enabling Application Security. *Int. Conf. on Pattern Languages of Programs Conference (PLOP)*. Monticello, Illinois, USA.

ENDNOTES

1. Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. Official Journal of the European Communities of 23 November 1995 No L. 281 p. 31
2. http://www.hipaa.org
3. http://www.glba.org
4. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification. W3C Working Group Note, 2006 - http://www.w3.org/TR/P3P11/
5. OMG. Unified Modeling Language: Infrastructure, 2007. Ver. 2.1.1, formal/2007-11-04 and OMG. Unified Modeling Language: Superstructure, 2007. Ver. 2.1.1, formal/2007-11-02 - http://www.omg.org/spec/UML/2.1.2/
6. Decreto Legislativo n. 196, 30 Giugno 2003, Codice in materia di protezione dei dati personali, Gazzetta Ufficiale n.174 del 29-7-2003 - Suppl. Ord. n. 123.
7. http://web.mit.edu/kerberos