A Case Study in Access Control Requirements for a Health Information System

Mark Evered
Serge Bögeholz

School of Mathematics, Statistics and Computer Science
University of New England
Armidale, 2351, NSW, Australia

Email: {markev,serge}@mcs.une.edu.au

Abstract

We present a detailed examination of the access constraints for a small real-world Health Information System with the aim of achieving minimal access rights for each of the involved principals. We show that, even for such a relatively simple system, the resulting constraints are very complex and cannot be expressed easily or clearly using the static per-method access control lists generally supported by component-based software. We derive general requirements for the expressiveness of access constraints and propose criteria for a more suitable access control mechanism in the context of component-based systems. We describe a two-level mechanism which can fulfil these criteria.

Keywords: access control, component, health information system

1 Introduction

The development of middleware technology has led to the practice of constructing software systems as collections of heterogeneous distributed components. Such systems are increasingly used for database integration, decision support systems, electronic commerce and many other applications. In general, the information stored within the components of these systems is sensitive and requires some form of access control. This is particularly important as the internet is increasingly used as the basis for distributed systems and as the threat from hackers and malicious software continues to grow. As well as protecting a system from these external threats, the access control should also ensure compliance with privacy laws and ideally, should guarantee that each user with access to the system uses the information only exactly as required for their role within the organisation.

Despite the sensitivity of the data and the growing threat, relatively little attention has been paid to the complexities of real-world access constraints in middleware development. Systems such as OMG's Corba (Blakley 2000) and Sun's Enterprise Java Beans (Hartman 2001) include a form of access control list (ACL) but these tend to be add-on features which remain limited and inflexible. Much attention has been given to encryption techniques but, while encryption is certainly important, it protects only the communication and authentication in the system. It provides only the basis for a secure access control mechanism.

In this paper we present a case study of the access control requirements for a Health Information System in a small aged care facility in rural New South Wales. The aim is to specify precisely the minimal access requirements for each of the involved principals and to use this as a basis for assessing how well the constraints can be expressed and enforced with existing middleware technologies.

We have chosen to investigate a facility which has strict procedures and policies in place for compliance with privacy laws but which, as yet, uses no computer system at all. All information is currently held on paper in filing cabinets with access physically controlled by the manager of the institution. The advantage of choosing a paper-based facility for the case study is that no compromises have been made to adapt the access constraints to the limitations of a computer system.

We begin in the next section by briefly reviewing the state of the art in access control for component-based systems. Section 3 describes the aged care facility, identifies the data and principals involved and describes the access constraints for each of the principals. In section 4 we formulate some general kinds of access constraint which can be derived from the case study and in section 5 outline some requirements for a mechanism which is able to express and enforce these kinds of constraints in a clear way. Section 6, gives an overview of and comparison with related research.

2 Access control in component-based systems

Traditionally, access control has been expressed in terms of read/write access. This is true for the files stored in a standard file system and also for the attributes associated with a record in a database system. An object-oriented approach can be used to support a finer-grained, higher level form of access control than this.

A component in a distributed application can be viewed as an object containing (possibly persistent) data, hidden by encapsulation and accessible via interface methods. This allows the access control to be expressed in terms of the interface methods of the object, which can be meaningful, application-level operations associated with the object in the real world. So, for example, we may define a persistent object which implements patient records in a Health Information System with methods for adding a new patient, adding a treatment for a patient, etc. (Fig. 1).

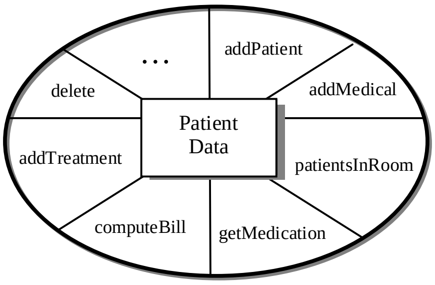

Fig.1 A persistent object for patient information

Note that this object contains the data for a set of patients rather than representing the data of a single patient. So, for example, the *setRoom* method has a parameter indicating which patient is meant and a parameter giving the room number for that patient. This concept of a container object is important for a number of reasons. One reason is that some operations involve more than a single patient. An example here is the *patientsInRoom* method which returns a list of the patients who are sharing a particular room. Another reason has to do with persistence. It is not practical to make each individual patient record into an independently persistent object. In fact often this not even desirable since the data may actually be stored in a database system with the object serving as a wrapper which hides the implementation and offers the high-level operations. Finally, for security and consistency, we want to ensure that no reference to data for a single patient can be retained and manipulated by a client independently of access via the container.

The access rights to this *Patients* object can be granted on the basis of the roles of the principals within the organisation. Software running on behalf of an administrative assistant at a hospital may, for example, have the right to invoke the *addPatient* method and the *patientsInRoom* method. For a nurse, access to the *getMedication* method would be allowed while only a doctor would have access to the *addMedical* method to add an entry to a patient's medical records. An X-ray technician would be allowed to invoke the *addTreatment* method which adds information both to the medical information and the billing information of a patient.

This idea of limiting access on the basis of interface operations goes back as far as Jones' and Liskov's (1978) suggestion of a static type-based constraint mechanism and Keedy's and Richards' (1982) semantic operations for persistent file objects. Some form of method-based access control has been incorporated into most contemporary middleware mechanisms. The COM+ security mechanism offers 'per-method access control lists' which record, for each method, a list of users allowed to invoke that method (Eddon 1999). In Sun's EJB, a set of roles can be defined in a deployment descriptor together with a list of the methods of the bean which can be invoked for each role (Hartman 2001). In the Corba Security Service there is an additional level of indirection with principals mapped to attributes, attributes mapped to rights (for a certain domain) and rights mapped to interfaces or interface methods (Beznosov 1999).

These mechanism clearly offer a finer-grained access control than a simple read/write approach but the question nevertheless remains as to how well they support the kinds of access constraint required in real applications. It is easy to present a simplified model as in the above diagram, but the definition of such an object for the requirements of a real system is considerably more complex.

3 A simple application

3.1 Overview

The case study is based on a Health Information System for an aged-care facility in rural New South Wales, Australia. The facility offers single room accommodation for some 30 resident. Administrative duties including the formulation of procedures and policies are handled by a manager who was the primary source for the case study information. Residents are cared for by health care staff, a nurse and volunteers with visits by doctors and, where necessary, physiotherapists.

The facility currently uses only paper records. The information stored includes personal, financial and medical information about each resident and contact details for the staff and visiting professionals. For the sake of simplicity, we have chosen to ignore the administration of staff salaries and other financial aspects of the institution in order to concentrate on the processing of the data concerning the care of the residents.

The proposed electronic version of the system is intended to completely replace the use of paper, providing intranet access to the HIS information stored on a server system. One PC will be used by the manager and a number of PCs will be available for use by the health care staff. Doctors and physiotherapists will make use of mobile devices which hook into the network dynamically.

The aim in formulating the access contraints for the various principals is to maintain at least the strictness of the current paper-based system and, ideally, to achieve a strict need-to-know access scheme. Before proceeding to the access rules, however, it is necessary to describe more fully the data which needs to be stored for each resident.

3.2 Data

There are two basic kinds of data maintained for each resident. The first is (relatively) static data entered into the system when a resident is admitted. This includes personal details such as name, sex, religion etc., medical insurance information and past medical records.

One important sub-category of this initial data is the **emergency details**. As well as medical information such as blood group, allergies etc., and contact details for the resident's doctor, this includes the contact details of a *responsible person* (formally referred to as next-of-kin) who is to be contacted in emergencies and who, if the resident is not mentally capable, can make decisions and provide signatures on behalf of the resident. Currently, a special card system is used to make this information available very rapidly.

Before admittance a checklist of this initial information and a legal agreement must be signed by the resident (or the responsible person). In the electronic system, these signatures are to be realised by a method invocation (represented at the user interface by a mouse click on an acceptance form).

The second kind of resident information is that which is used and updated in the normal day-to-day running of the facility. Most of this is represented by three sets of information:

The care plan

This is a working document that contains detailed information and instructions regarding the day-to-day care of the resident, eg. what assistence is required with meals, hygiene etc. A care plan is started for each resident on admission and is updated on a regular basis. Old versions of the care plan are archived.

Progress notes

These are observational entries covering such aspects as physical mobility, appetite, behaviour, mood and the general state of the resident. Progress notes are used to update the care plan. Progress notes older than one year are also archived.

Medical records

A number of different doctors visit the facility with one doctor visiting each week on 'clinical day'. Residents can choose which of these doctors they wish to attend them. The facility requires that each resident undergo a medical examination at least every six months and medication is reviewed at least every three months. After each examination the doctor adds an entry to the medical records of the patient. Currently these records are kept in duplicate with one copy in the facility and another kept by the appropriate doctor. The doctor's copy additionally includes **private notes** for each of his/her patients. In the electronic version this duplication is to be eliminated.

In the current paper-based system, medical entries older than one year are archived and filed away in a locked room. Recent medical entries are stored in a locked filing cabinet in a more accessible location with access controlled by the manager.

3.3 Access rules

Manager

The manager has the broadest access to the information, including access to personal, financial, clinical and medical information about each resident. This does not mean, however, that she has unrestricted access. Although she can enter the past medical records when a resident is admitted, she cannot subsequently add medical entries to the system. In the current paper-based system, this is prevented by requiring that any new entry be signed by the doctor making that entry. Also, she cannot view the private notes of doctors and clearly she cannot sign the legal agreement on behalf of a resident.

Only the manager is allowed to add a new resident to the system and to start or update the care plan of a resident. The care plan is updated in consultation with the resident or the responsible person.

Only the manager is allowed to delete the information about a resident but here also that right is restricted. Privacy laws require that the information be held for a certain period after a resident leaves the facility. This period is seven years for someone who is not of Aboriginal or Torres Strait Islander descent and nine years for someone who is.

Health Care Workers

The health care workers are required to sign a confidentiality agreement before they have access to any resident data. Their main form of access is to view the care plan for each resident and to add progress note entries based on their observations. Access to emergency details is available for all staff.

Health care workers can view recent medical records of residents (up to one year old) but cannot normally view older medical information. For a special purpose, access to an older medical record can be sought and obtained from the manager.

Because of the physical access control in the current paper system, the manager has an overview of who has accessed what information. In an electronic system this overview must be supplied by some form of logging of accesses and access attempts. Clearly, the manager does not wish to be informed about every access but some logging is still necessary. So, for example, the manager should be made aware of repeated attempts by a health care worker to access information beyond their rights.

Doctors

A visiting doctor has access to all the medical information of residents who are his/her patients and can add entries to their medical records. Doctors can also add private notes about a resident, which, on the basis of doctor-patient confidentiality, are not visible to health care staff or the manager. Doctors need not sign a confidentiality agreement since they are bound by a code of professional conduct.

Occasionally, due to pressing circumstances, it may be necessary for a visiting doctor to examine a resident who is not normally his/her patient. This is permitted with the consent of the resident or the responsible person and the notification of the manager. The doctor then has temporary access to the resident's medical records (but not to the private notes of the resdient's usual doctor).

Residents

The principals involved in an information system are usually thought of as staff members of a certain organisation. Privacy laws require, however, that a person should have full access to any information stored about them (unless the well-being of a third party would be jeopardised by revealing the information).

In our case study this means that the residents themselves must also be regarded as principals for whom we define access rights. In the current paper system, all the information stored about a resident is accessible to the resident by arrangement with the manager. The need to make this arrangement is only to allow the manager and resident to set a time that is convenient to both of them.

In an electronic system we can allow residents to hook into the network at any time as long as they can view only their own data and cannot make arbitrary changes. This includes the right to view even the private notes entered by a doctor.

Others

For the sake of brevity we will not discuss in detail the access rights of all the other principals. These include the nurse, visiting physiotherapists, volunteer helpers and the responsible person for each resident. In a larger institution some of the duties of the manager would be performed by administrative staff but in this facility, such staff are not necessary.

For a full discussion of principals and the corresponding access rules see (Bögeholz, 2003).

4 General access control requirements

4.1 Methods of the *Residents* object

In this section we discuss how the resident information described above can be stored in a persistent object with appropriate interface methods providing the operations required at the application level. We then proceed to describe a number of basic kinds of access constraint which need to be imposed on the invocation of the methods in order to enforce the access rules described informally in the previous section.

The following Java interface type is not the full definition of the *Residents* object. Rather it is intended to give an indication of the nature of the interface and to list example methods which are relevant to the subsequent discussion of access control. The full interface definition of the *Residents* object and other objects comprising the HIS system, together with a description of their implementions are given in (Bögeholz, 2003).

```
public interface Residents {
  int newResident();
  void setName(int key, String name);
  void setDescent(int key,
                  boolean aboriginalTorres);
  void setDoctor(int key, int doctorId);
 int getDoctor(int key);
  void giveConsent(int key);
 void addMedicalTreatment(int key,
                          String treatment,
                          int enteredBy);
  String getMedicalTreatment(int key,
                             int index);
  void adjustCarePlan(int key,
                      String carePlan);
  String getEmergencyDetails(int key);
  ...
}
```

The *newResident* method adds a new resident record to the object and returns a key for that record. The key is given as a parameter in many of the other methods to identify which resident is intended. Some *String* parameters and results are in XML format to provide a standard way of passing structured information.

The *addMedicalTreatment* method adds an entry to the list of medical treatment entries for a resident. It has a parameter for indicating who is making the entry. The *getMedicalTreatment* method can be used to retrieve this information. An *index* value of 0 means the most recent entry, 1 means the previous entry etc.

4.2 Access to a subset of methods

This is the fundamental kind of access restriction in an object-oriented approach to access control. So, for example, nobody but the manager should be allowed to invoke the first four methods shown above and only a doctor should be able to invoke the *addMedicalTreatment* method.

A particularly interesting use of this kind of restriction is in realising signatures. The *giveConsent* method is invoked by a resident to register consent to the legal agreement with the aged care facility. If the access control mechanism can guarantee that only the resident can invoke this method, then the fact that it has been invoked can be regarded as a signature. The same mechanism can be used by health care staff for signing the confidentiality agreement.

4.3 Access dependent on attribute value

Sometimes the question of whether an access should be allowed to proceed depends on the information stored for a particular resident. So, for example, the manager is only allowed to invoke the *delete* method for a resident if the values stored for 'date of leaving facility' and 'descent' indicate that privacy laws would permit that deletion.

Similarly, a doctor may (normally) only invoke the *getMedicalTreatment* method for a resident for whom that doctor has been stored as the resident's chosen doctor. In a larger institution we might want to restrict the access of health care staff to information of only those residents in the wing/ward/department to which a member of staff is assigned.

4.4 Access with time constraints

A visiting doctor can treat a resident who is not normally his/her patient if there is a pressing reason and the resident agrees. This access right should not be permanent, however. It is sufficient if the doctor can access the relevant information on the day of the examination. The access right should subsequently expire.

The same situation exists for a health care worker who has been granted permission to retrieve medical records older than one year. This permission is for a special reason and should be limited in time.

4.5 Access based on call history

Like the constraints in 4.3 above, this kind of constraint depends on the state of the object but here the relevant factor is not the attributes of the information being stored but the allowed sequence of method invocations. So, for example, a health care worker cannot invoke any method to retrieve information about any resident unless he/she has already invoked the method for signing the confidentiality agreement.

A special case of access based on call history is the permission to invoke a method only once. The manager should not be able to re-set the 'descent' information once the method to set this value has been invoked and the value signed-off by the resident. Similarly, the invocation of the method to sign the legal agreement should only be invoked once by a resident.

4.6 Access with fixed parameter value

A resident is allowed to invoke any of the methods which return information stored about him/her but, of course, has no access to information stored about other residents. This implies that a resident only be allowed to invoke methods with a particular value of the *key* parameter.

Similarly, each doctor must be restricted to passing a certain value for the *enteredBy* parameter of the *addMedicalTreatment* method, namely, the value which represents the identity of that doctor.

4.7 Access with logging

Sometimes it is desirable to keep a record of who has done what and when with a persistent object. In a high-security context it may even be necessary to record every invocation of an object but in most information systems this would result in a flood of data which would tend to hide rather than expose the accesses of interest.

In this case study, the manager indicated that she would wish to be informed about certain invocations of the *Residents* object by health care workers and volunteers. So, for example, a record must be kept of exactly when and by whom emergency details for a resident have been retrieved.

Logging may be required both for successful and for unsuccessful method invocations. The manager also needs to be informed if a member of staff repeatedly attempts to perform some operation on the object for which they have no permisison.

5 Towards an ideal access control mechanism

5.1 Criteria

Clearly, a single case study is not sufficient to motivate the design of an access control mechanism. Nevertheless, it is instructive to consider what such a mechanism would need to offer to cater for even such a small example system. As well as being able to express the kinds of access control listed in the previous section, we claim that an ideal mechanism must also fulfil a number of further generic security criteria:

Concise

Access control is no use if it is not correct. If it is possible to express complex constraints only in an awkward or long-winded way, then errors are likely to be made. Our first criterion is therefore that the mechanism should allow the constraints to be expressed in an easy, concise way. This is especially true for the most common kinds of constraints. In general, mechanical repetition or extraneous effort in the expression of the rules are indications that something is amiss.

Clear

Closely related to concise expression is the clarity of the mechanism. Access constraints must be not only expressed but also checked. Ideally, at least for the most common cases, it should be apparent at a glance what the rule is saying and therefore whether it is correct.

Aspect-oriented

A third criterion is that the access control information should be separated from the application code, not embedded into or mixed up with it. This corresponds to the idea of aspect-oriented programming (Kiczales 1997) where separate aspects of a program such as security and synchronisation are formulated separately and then combined automatically by an 'aspect weaver'.

This separation makes both the application code and the access control easier to understand and also allows the same application object to be used in different security contexts with different access rules.

Fundamental

Ideally, an access control mechanism is integrated at a fundamental level within a system. For a component-based system this means that the access control should be integral to the middleware rather than an optional add-on. One advantage of this is that developers are forced to address access control questions from the very start. Another advantage is that it is then more difficult for hackers to 'get around' the mechanism.

Positive

The access rights of a principal should be expressed in terms of what that principal *is* allowed to do rather than what he/she *is not* allowed to do. This ensures that the default is that no access at all is allowed and that each permission must be explicitly listed.

Need-to-know

A strict need-to-know approach to access control is not only desirable in military environments. Financial transactions and the manipulation of sensitive personal data are increasingly being perfomed electronically via networks and users have a right to expect that no more of their data is being revealed than is absolutely necessary for a particular service.

Efficient

Clearly, a certain overhead will always be involved in performing access control checks but this overhead should be kept at a reasonable level.

5.2 Realisation

This case study demonstrates that access constraints in the real world, if taken seriously in terms of the need-to-know principle, can be very complex, even for a small information system. Standard access control mechanisms for component-based systems are based on a static per-method access control list and cannot easily support these complex kinds of constraints. It is possible to add more methods or more objects to the system in an attempt to achieve the desired result but if this is done in an ad-hoc way then many of the criteria listed in section 5.1 will be violated and, as a result, errors and loopholes will be introduced.

This is reminiscent of the attempt to use *path expressions* to specify the synchronisation constraints for objects. Here also, the constraints are expressed solely in terms of method names. For simple cases this is sufficient and even elegant but for realistic examples such as reader/writer synchronisation with priorities, the only solution is to introduce additional methods which have nothing to do with the application code (Habermann 1974).

One way to enable complex access control constraints while keeping them separate from the application code is to use some form of *security proxy*. All method invocations to an object are passed through this proxy and checked before being passed on to the underlying object. Such a security proxy still has a number of problems in terms of the above criteria, however. One problem is that the clarity is hindered by the need to program all the constraints for all of the principles in a single object. Another problem is that all of the methods are still visible to all of the principles even if they are not invocable. This violates a strict need-to-know approach and complicates the use of the object. This problem is magnified by the fact that an object in a real system may easily have over 100 methods on its interface. Thirdly, such a proxy mechanism is not generally an integral part of the component architecture but rather an add-on feature (if available at all).

In order to fulfil all the criteria we propose a two-level approach. A flexible low-level mechanism can guarantee the security and efficiency of the approach while a high-level language construct can enable conciseness and clarity to support correctness.

5.2.1 Bracket capabilities

Our approach for the lower level is the *bracket capability* mechanism. The concept behind this mechanism is presented more fully in (Evered 2002a) and an implementation in the **Opsis** system is described in (Evered 2002b). The mechanism is based on object capabilities rather than ACLs because capabilities enhance and simplify security by unifying object naming with the protection mechanism (Wilkes 1979). Our capabilities differ from traditional sparse capabilities in that they contain not an object identifier, but an identifier for a (capability) server which knows the location of the object. This indirection allows for object migration. When a persistent object is created, a capability for the object is created and registered with the capability server. The capability server also contains other information about accessing the object, including the type of the object as seen by the possessor of that capability.

To gain access to an object, the object is ‘opened’ using a capability. For example:

```
Residents res = (Residents) c.open();
```

where `c` is a variable of type `Capability`. The object can then be accessed via invocations of its interface methods. For example,

```
res.setName(12345, “Smith, John”);
```

So far, this is not much different from other mechanisms based on object capabilities. The main difference is seen when the possessor of a capability wishes to grant a more restricted view of the object to other users in the system. This is done by a call to the **`refine`** method. Each persistent object, as well as implementing an interface such as **`Residents`** also implements the standard interface **`Persistent`** which includes methods such as **`deleteObject`**, **`deleteCapability`** and **`refine`**. The **`refine`** method is called as follows:

```
x = (Persistent) c.open();
Capability cref = x.refine(interface, class);
```

where **`interface`** denotes the type with which the persistent object is to be viewed (when opened using the capability **`cref)`** and **`class`** denotes the class of an object *through which calls to the persistent object will pass* when invoked via **`cref`**. The result of the **`refine`** call is depicted in Fig. 2.

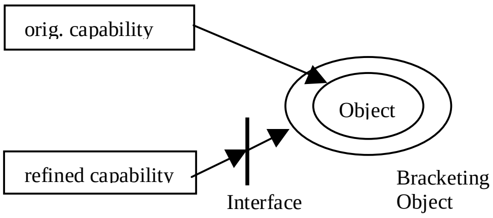

Fig 2: The result of the 'refine' operation

It can be seen that, as well as having a restricted interface, calls using the capability **`cref`** are directed through a *bracketing* object. This bracketing object is stored together with the underlying component in the same way that access rights are stored with the component for other sparse capability mechanisms.

A copy of **`cref`** can be given to the principals who are to have this kind of access. By creating as many such new capabilities as we need, we can give each principal exactly the interface and the bracketing code required.

So, for example, we can achieve the same effect as standard per-method ACLs by listing only certain methods in the interface. In the case of bracket capabilities, however, the missing methods are not even visible to the principal. Additionally, we can achieve more complex access constraints by providing whatever checks are necessary in the code of the bracketing object.

We can compare this approach to the simple per-method approach of Corba and EJB in which more complex constraints are generally programmed in the code of the component to be protected (Table 1).

As can be seen, the use of bracket capabilities can fulfil many of the criteria but the access constraints are still written as normal program code which is often difficult to understand as well as tedious and repetitious and this in turn is likely to lead to errors. For this reason, we propose, as the second level, a special language construct for formulating access constraints.

5.2.2 A language construct for access constraints

In order to fulfil the requirements of conciseness and clarity, we define an 'access construct' which provides a high-level description of the access constraints and is automatically translated into the class to be used as the bracketing class for a particular capability. For the most complex cases, this construct can contain arbitrary program code, but for the most commonly occurring cases, such as described in section 4, it is designed to be as clear as possible.

Criterion	Bracket Capabilities	Simple Per-method ACLs
Concise	poor	poor
Clear	poor	poor
Aspect-oriented	good	poor
Fundamental	good	poor
Positive	good	good
Need-to-know	good	fair
Efficient	good	good

Table 1: Bracket capabilities vs standard approach

So, for example, the access given to a doctor will include the following:

```
interface DoctorView {
   String getMedicalTreatment(int key,
                                int index);
  ...
}
```

```
access DoctorAccess to Residents
  provides DoctorView {
  int doctorId;
  pre { check(doctorId==getDoctor(key)) }
  ...
}
```

where the interface defines the doctor's view of the object and the access construct defines further constraints. In this case the constraint is that the doctor associated with this patient is the same as the doctor making the method invocation.

A formal definition of the access construct is beyond the scope of this paper. Informally, it is equivalent to a Java class definition with the addition of a `pre` and/or a `post` section for defining actions to bracket a method invocation. It can also contain variable values to be substituted for parameters of the methods of the bracketed object. This is adequate for clearly expressing all the constraints of the case study without awkwardness or repetition.

Given this second level on top of the bracket capability mechanism, the comparison with the standard approach is as shown below (Table 2).

Criterion	Access Construct	Simple Per-method ACLs
Concise	good	poor
Clear	good	poor
Aspect-oriented	good	poor
Fundamental	good	poor
Positive	good	good
Need-to-know	good	fair
Efficient	good	good

Table 2: Access construct (on top of bracket capabilities) vs standard approach

6 Related work

As mentioned above, standard middleware systems such as Corba, COM+ and EJB include the possibility of a per-method, role-based access control list for limiting the access of principals to interfaces or objects. In some cases, fixed forms of rule-based access, such as access at certain times of day, are supported. These correspond only to simple, special cases of access control. No direct equivalent of the complex restrictions required for the case study are supported. No direct equivalent of a restricted view of the object is supported for hiding the existence of unallowed methods and parameters from the principals. In both of these middleware technologies, the use of ACLs instead of capabilities makes the security mechanism an add-on feature rather than fundamental and detracts from the security.

Object capabilities have been used in a number of research systems, most notably the Monads system (Rosenberg and Abramson, 1985) and for the 'protected subsystems' in the Multics system (Saltzer, 1973) but these capabilities require architectural support (or at least a special operating system kernel) and so are not appropriate for heterogeneous networks. Brose (1999) has proposed a language-based extension to the Corba security model in which the allowed 'views' for each user are defined in terms of the methods of an object type. Like the Monads and Multics systems and the ACL approaches of Corba and EJB, however, these support only simple per-method access control. In all cases, all of the methods are visible to all principals even if they may not be invoked and parameters cannot be fixed to certain values.

The concept of 'bracketing' for applying access constraints has been suggested both as a programming language construct (Keedy et al., 2000) and as a form of 'design pattern' (Gamma, 1995). The suggested programming language approach is interesting in supporting the *reuse* of the bracketing code but it does not allow modification of the interface to the underlying object or a concise and clear expression of access rules.

One use of the *proxy* design pattern is as a protection (or access) proxy. In this case, the interface is identical to the underlying object. The proxy decides whether the access can proceed and returns an error if it should not. Bracketing objects which modify the interface offered to a client cannot be seen as strict proxies. They can be seen as special cases of the adapter pattern but whereas an adapter is usually used to provide the view the client would *like* to have of the underlying object, in these cases the adapter is providing the view the client is *allowed* to have.

The concept of providing a user with a restricted view of persistent data is reminiscent of database systems. Traditional database views are attribute-oriented and not method-oriented, however, and therefore cannot support the flexible kinds of access control required for our example. This attribute-orientation is true even for most object-oriented databases (Mishra and Eich, 1994). Notable exceptions are the method-based model of Fernandez, Larrondo-Petrie and Gudes (1993) and the CACL system of Richardson, Schwarz and Cabrera (1992). The former provides an 'Execute' access right for invoking a method of a persistent object. This is similar to the per-method access control of contemporary middleware systems. The latter supports the concept of an 'authorization type' as a restricted view of an object but does not allow parameter constraints, state-dependent constraints etc. to be specified as part of the view.

7 Conclusion

Per-method object-based access control allows access restrictions to be expressed in terms of high-level, application-relevant operations on the components of a software system. This is a considerable improvement on the data-oriented read/write access restrictions in file systems and in most database systems. Nevertheless, the question arises whether the access constraints of real information systems can adequately be expressed merely by specifying a method subset for each principal to each interface or object of the system.

We have presented a case study of the access constraint requirements of a very simple real-world health information system. The information system involves the resident data for a small aged care facility and the access to be permitted for staff members, for visiting professionals and for the residents themselves. We have chosen a system which is currently still paper-based so that the access procedures and policies have not been influenced by the limitations of contemporary information systems software.

The case study shows clearly that the access constraints for even such a simple system, if expressed in terms of the minimum access required for each principal, are extremely complex. We conclude that the per-method access control lists of standard component technology are not adequate for expressing such real-world access constraints in a clear, concise manner.

We have categorised the kinds of access rules required for the case study and on the basis of these and further generic criteria, have formulated some requirements for an ideal access control mechanism. We propose a form of access adapter which combines the expressiveness of general program code with the clarity of a declarative approach for the most common cases.

Ongoing work includes further case studies to investigate the adequacy of the access construct as currently defined and work on a Java implementation of the mechanism.

References

Beznosov, K., Deng, Y. (1999): A Framework for Implementing Role-based Access Control using CORBA Security Service, *Proc. 4th ACM Workshop on Role-based access control*, Fairfax.

Bögeholz, S. (2003): *Access Control in a Distributed Health Information System: A Case Study*, Masters Thesis, University of New England, Armidale.

Blakley, B., Blakley, R., Soley, R.M. (2000): *CORBA Security: An Introduction to Safe Computing with Objects*, Addison-Wesley.

Brose, G. (1999): A View-Based Access Control Model for CORBA, in: Jan Vitek, Christian Jensen (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, Springer.

Eddon, G. (1999): The COM+ Security Model Gets You Out of the Security Programming Business, *Microsoft Systems Journal*, November.

Evered, M. (2002): Bracket Capabilities for Distributed Systems Security, *Proc. 25th Australasian Computer Science Conference*, Melbourne.

Evered, M. (2002): Opsis: A Distributed Object Architecture Based on Bracket Capabilities, *Proc. Conference on Technology of Object-Oriented Languages and Systems*, Sydney.

Evered, M. (2003): Flexible Enterprise Access Control with Object-oriented View Specifications, *Australasian Information Security Workshop*, Adelaide.

Fernandez, E.B., Larrondo-Petrie, M.M., Gudes, E., A. (1993): Model of Methods Access Authorization in Object-oriented Databases, Proc. of the 19th VLDB Conference , Dublin.

Gamma, E. et al. (1995): *Design Patterns*, Addison-Wesley.

Habermann, A.N., Campbell, R.H. (1974): The specification of process synchronization by path expressions, *Lecture Notes on Computer Science*, 16.

Hartman, B., Flinn, D.J., Benznosov, K. (2001): *Enterprise Security with EJB and CORBA*, Wiley.

Jones, A., Liskov, B. (1978): A language extension for expressing constraints on data access. *Communications of the ACM*, 21(5):358-367, May.

Keedy, J.L., Richards, I. (1982): A Software Engineering View of Files, *Australian Computer Journal*, 14, 2.

Keedy, J.L., et al. (2000): Software Reuse in an Object Oriented Framework: Distinguishing Types from Implementations and Objects from Attributes, *Proc. Sixth International Conference on Software Reuse*, Vienna.

Kiczales, G. et al. (1997): Aspect-oriented programming, *Proc. European Conference for Object-Oriented Programming*, Finland (Lecture Notes in Computer Science, vol. 1241). Springer.

Mishra, P., Eich, M.H. (1994): Taxonomy of views in OODBs, *Proc. ACM Computer Science Conference*.

Richardson, J., Schwarz, P., Cabrera, L. (1992): CACL: Efficient Fine-Grained Protection for Objects, *Proc. OOPSLA Conference*.

Rosenberg, J., Abramson, D. A. (1985): The MONADS Architecture: Motivation and Implementation, *Proc. First Pan Pacific Computer Conference*, p. 4/10-4/23.

Saltzer, J.H. (1973): Protection and the Control of Information Sharing in Multics, *Symposium on Operating System Principles*, Yorktown Heights, NY.

Wilkes, M.V., Needham, R.M. (1979): *The Cambridge CAP Computer and its Operating System*, North Holland.