Person Identification Service (PIDS) Specification

April 2001
Version 1.1

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form at *http://www.omg.org/library/issuerpt.htm*.

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

OMG Documents

In addition to the CORBA Core Specification, OMG's document set includes the following publications.

OMG Modeling

The Unified Modeling Language (UML) Specification defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems. The specification includes the formal definition of a common Object Analysis and Design (OA&D) metamodel, a graphic notation, and a CORBA IDL facility that supports model interchange between OA&D tools and metadata repositories. The UML provides the foundation for specifying and sharing CORBA-based distributed object models.

The Meta-Object Facility (MOF) Specification defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models. The MOF provides the infrastructure for implementing CORBA-based design and reuse repositories. The MOF specifies precise mapping rules that enable the CORBA interfaces for metamodels to be automatically generated, thus encouraging consistency in manipulating metadata in all phases of the distributed application development cycle.

The OMG XML Metadata Interchange (XMI) Specification supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information. The specification supports the encoding of metadata consisting of both complete models and model fragments, as well as tool-specific extension metadata. XMI has optional support for interchange of metadata in differential form, and for metadata interchange with tools that have incomplete understanding of the metadata.

Object Management Architecture Guide

This document defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

OMG Interface Definition Language (IDL) Mapping Specifications

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

CORBAservices

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include Collection, Concurrency, Event, Externalization, Interoperable Naming, Licensing, Life Cycle, Notification, Persistent Object, Property, Query, Relationship, Security, Time, Trader, and Transaction.

CORBAfacilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include Internationalization and Time, and Mobile Agent Facility.

Object Frameworks and Domain Interfaces

Unlike the interfaces to individual parts of the OMA “plumbing” infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren’t required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the *Common Object Request Broker: Architecture and Specification*, *Interworking Architecture* chapter.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal (published) specifications are available from the OMG website *http://www.omg.org/technology/documents/formal/index.htm*. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- 2AB
- 3M Health Information Systems
- Ardent Software, Inc.
- Care Data Systems, Inc.
- CareFlow/Net, Inc.
- FUJITSU LIMITED
- HBO & Company
- HealthMagic, Inc.
- HUBlink, Inc.
- IBM Corporation
- IDX Systems Corporation
- INPRISE Corporation
- IONA Technologies PLC
- Oacis Healthcare Systems
- Object Design, Inc.
- Objectivity, Inc.
- Oracle Corporation
- Persistence Software, Inc.
- Protocol Systems, Inc.

- Secant Technologies, Inc.
- Sholink Corporation
- Sun Microsystems, Inc.
- Versant Object Technology Corporation

Overview 1

Contents

This chapter contains the following topics.

1.1 Introduction

Throughout a person's lifetime he or she may have episodes of care provided by dozens or hundreds of healthcare providers, most of whom may be assigning and maintaining patient IDs autonomously. In this arrangement, each organization simply assigns IDs that uniquely identify patients within its local ID Domain of ID values, with the result that these ID values are meaningless outside that system or organization. These autonomously managed IDs suit the purposes of recording and retrieval of service records for the local organization. However, there is no basis for efficient collection or correlation of health records among multiple venues.

A typical healthcare information system will permit the user to submit a search for a person’s record using some combination of identifying parameters for the person. When the user must collect a patient's healthcare information from a different organization or from a disparately-keyed system in the same organization, s/he typically must perform a new search in that other system - or ask a medical records person in the other organization

to perform the search - in order to identify the person and retrieve the needed information.

In recent years, changes in the business of healthcare have made it both increasingly important and increasingly difficult to access the continuum-complete record of care for an individual. Risk-shared and capitation-based reimbursement policies have made it absolutely necessary to avoid redundant treatments. Increased specialization of providers has caused increased fragmentation and distribution of patient records.

Finally, organizational consolidation, growth and flux are exacerbating the problems associated with managing IDs. Most large integrated delivery systems are now competing on the basis of population share.

1.2 Scope

When identifying a person there is a variety of identifying information that may be used including demographics (address, place of birth, etc.), biometric information (finger print, photograph, blood type) and IDs they may have from other sources (SSN, drivers license number, insurance number). Some biometric information (e.g., blood type or weight) may also be considered clinical information.

Question: Can't PIDS be used for accessing clinical information about a patient such as electronic medical records?

Answer: No. There is a fuzzy line between biometric information and clinical information but the distinguishing trait is their stability over time. The predefined names of traits do not cover clinical information and PIDS is not meant to store clinical information.

The information about a patient that is useful for identification purposes are those traits that remain constant or change very slowly over time. The expectation in using them is that they have not changed so they can be stored and then used to identify the person the next time they are encountered. The information being gathered as part of the clinical diagnosis is expected to have changed and is, in general, unstable over time.

The key word in Patient Identification Service is "Identification." Identity is definitely not unique to patients. When looking at the need for identification in many fields and looking at the interfaces being proposed, we envision this being applicable to much broader entities than just patients.

Question: Should the thing being identified in this specification be patients, persons, or subjects?

Answer: Persons

The concept of identifying a subject by its characteristics is applicable to more than just persons but there are a couple problems proposing this specification to cover them. We need to define a default set of identifying information that can be used with this service for identifying patients. This information is specific to identifying persons (not just patients) but not general entities or subjects.

Applying this specification to general entities that need to be identified is beyond the scope of this specification and even the CORBA Healthcare DomainTask Force.

Question: Isn't a Person Identification Service needed in other domains than just healthcare? Why is CORBA Healthcare standardizing it?

Answer: Yes, it is needed in other domains - but as an ancillary part of work flow as their real area of concern is finance, telecommunications, manufacturing, etc. In healthcare, the person (we call them patients) is what the business is all about.

Other domains need to identify a person to make sure a financial transaction or other business contract is applied to the correct person. In healthcare, it is important when diagnosing a problem that you have the same "physical" being that a patient record refers to. Therefore, incorrectly identifying the person can have more severe consequences.

For these reasons it makes sense that CORBA Healthcare standardize interfaces for identifying persons. These interfaces should be useful outside healthcare as well.

1.3 Design Goals

This specification defines the interfaces of a CORBA Person Identification Service (PIDS) that organizes person ID management functionality to meet healthcare needs. The PIDS is designed to:

- Support both the assignment of IDs within a particular ID Domain and the correlation of IDs among multiple ID Domain.
- Support searching and matching of people in both attended-interactive and message-driven-unattended modes, independent of matching algorithm.
- Support federation of PIDS services in a topology-independent fashion.
- Permit PIDS implementations to protect person confidentiality under the broadest variety of confidentiality policies and security mechanisms.
- Enable plug-and-play PIDS interoperbility by means of a "core" set of profile elements, yet still support site-specific and implementation-specific extensions and customization of profile elements.
- Define the appropriate meaningful compliance levels for several degrees of sophistication, ranging from small, query-only single ID Domains to large federated correlating ID Domains.

1.4 Domain Reference Model

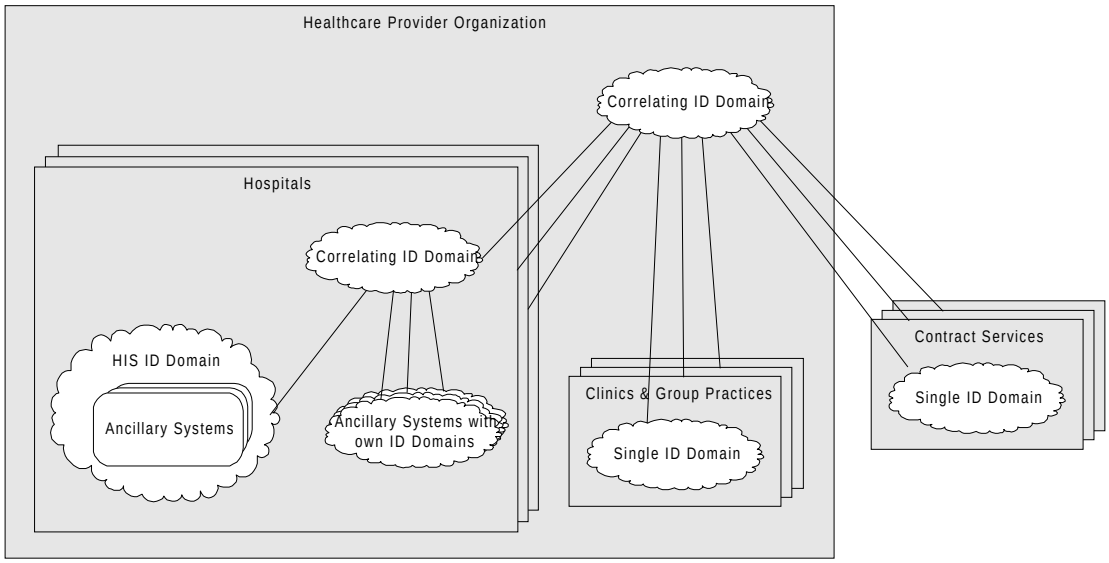


Figure 1-1 Domain Reference Model for PIDS

Figure 1-1 represents a reference model for healthcare as it relates to this specification. This is not an attempt to define a globally useful reference model for healthcare as this model was chosen to highlight the aspects of healthcare related to person identity.

A hospital is likely to have one primary ID Domain, which would typically be defined by the Healthcare Information System (HIS) or Admit Discharge Transfer (ADT) system. Many ancillary or departmental systems would be binding person-related data (demographic, clinical, financial) to IDs in that ID Domain. There may be other ancillary systems that use their own identification to make them each a separate ID Domain. The hospital may manually mirror these ID Domains against a “main” ID Domain or electronically connect these systems to a common registration process to actually consolidate them into a single ID Domain. Today they use an MPI system to do this.

Today Healthcare Provider Organizations may consist of multiple hospitals and multiple clinics forming an Integrated Delivery System. Each clinic that is automated will likely have one practice management system, which manages its own ID Domain.

A Healthcare Provider Organization also has contract services such as reference labs. Each contract service would have its own ID Domain. The Healthcare Provider Organization may need a higher level of correlation that consolidates the various ID Domains.

There could be yet higher levels of correlation among Healthcare Provider Organizations. These do not add new relationships as they are just correlating over correlating ID Domains as we have here. Their population sizes would potentially be much larger. Even though it is not shown in Figure 1-1, an ID Domain could have multiple correlating ID Domains that correlate its IDs with other ID Domains.

1.5 PIDS Conceptual Model

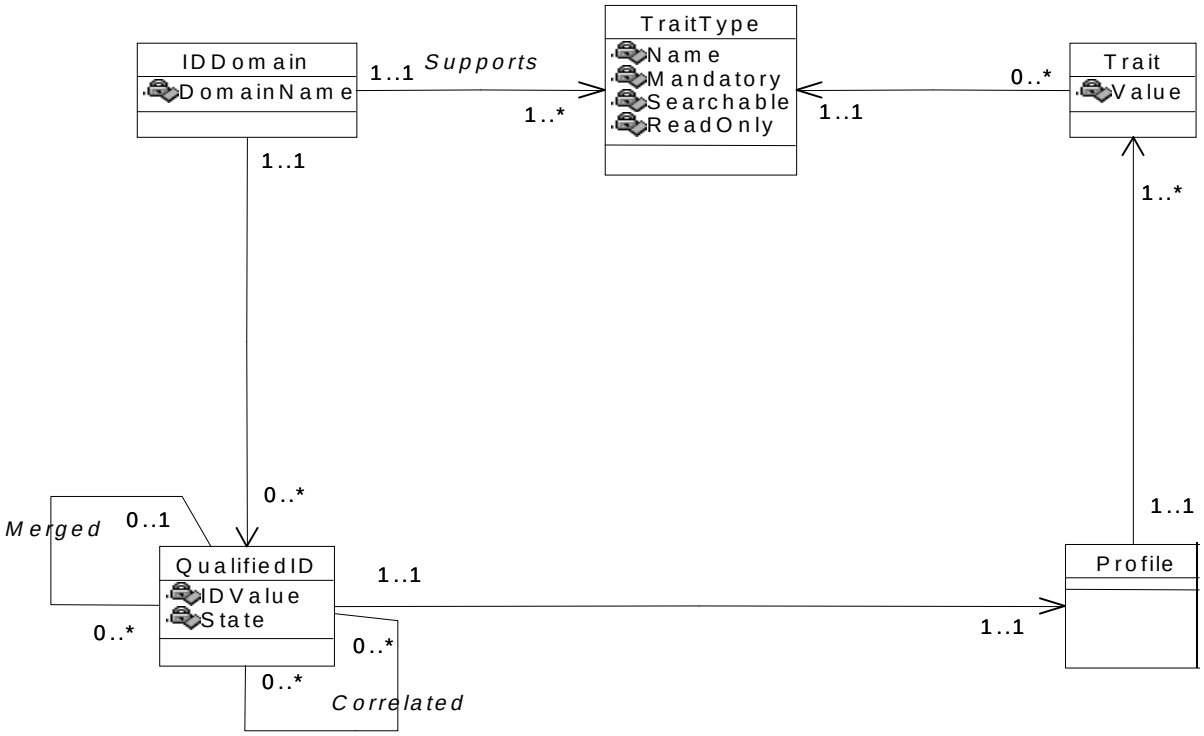


Figure 1-2 PIDS Conceptual Model Diagram

The PIDS Conceptual Model shows the core data elements that PIDS implementations must deal with. It also shows how these relate to a real world person. The following sections will describe this data and show how they are used by PIDS.

In PIDS implementations, the QualifiedID associations might carry administrative and auditing attributes such as, time stamp, user stamp, source system, and specific operation types for merge and correlation. For example, specific operation types could be added to correspond to the HL7 2.3 merge and link events.

1.6 PIDS Identification Model

The following figure provides the basic structural elements of our identification model:

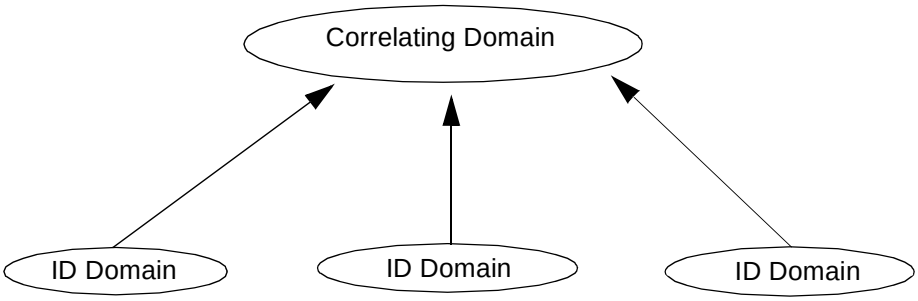


Figure 1-3 PIDS Identification Model Diagram

The ID Domain is the basic building block of our PIDS model. An ID Domain maintains a unique Identifier (ID) for each person identity to be represented in the ID Domain. Ideally there is one and only one ID per person, but in reality there may be duplicates where a given person has been assigned an additional ID in the same ID Domain. For internal consistency, the ID Domain cannot assign two persons the same ID or it would have no means to distinguish between the two entries. The ID is an internal control mechanism and may or may not be used externally. Thus the ID and its ID Domain together create a unique ID for the person.

The ID Domain provides a framework for representing existing systems. For example, the MPI in a registration system is an ID Domain as is the patient index in an ancillary system. In other words, ID Domains already exist in any patient oriented system. Our terminology simply provides descriptive unit for existing identification processes.

Within the PIDS specification, several interfaces are detailed. The two most directly used with an ID Domain are **IdentifyPerson** interface and **ProfileAccess** interface. The **IdentifyPerson** interface is basically a query used to send traits to be matched in the ID Domain and to receive the matching candidate(s). The **ProfileAccess** interface can be a query or an update used to send an ID for a specific person and to receive that person's profile. A profile is a set of traits containing the person's values for their respective traits. Two additional interfaces are specialized forms of the **ProfileAccess** interface, namely the **SequentialAccess** interface and the **IdentityAccess** interface.

The coordinating structural unit for the PIDS model is the Correlating Domain. The Correlating Domain allows access to the correlated profiles of the IDs in all of the participating ID Domains. This building block provides the framework for cross-referencing IDs across the participating ID Domains. The Correlating Domain is a new building block which complements existing ID Domains.

Two interfaces illustrate the role of the Correlating Domain. First, profiles are added and correlated via the **CorrelationMgr** interface (**PersonIdSeq** register). Second, cross-referenced IDs are obtained by the **CorrelationMgr** interface (**UniquePersonIdSeq get_corresponding_ids**).

Integrity of each domain is maintained by their respective managers – the **IDMgr** interface and the **CorrelationMgr** interface.

To illustrate the use of this model in actual implementations, consider the following examples.

The MPI in a registration system or the patient index in an ancillary system is an ID Domain. When a registration system is interfaced to ancillary systems, the registration systems MPI would be an ID Domain and the ancillary systems using the ID assigned by the registration system would be codependent domains. When multiple provider systems are interfaced, such as for an integrated delivery system, the enterprise MPI would be the Correlating Domain for the participating ID Domains of the provider systems. If the enterprise were to issue the **IdentifyPerson** interface and the **ProfileAccess** interface to the Correlating Domain instead of to each ID Domain, then the enterprise MPI would be both a Correlating Domain and an ID Domain.

The PIDS model provides a tool to manage identification, which provides flexibility for both the short-term (existing systems) and for the long-term (quickly accommodate business requirements). The PIDS model also allows both hierarchical and peer structures to evolve. Since a Correlating Domain can correlate both ID Domains and other Correlating Domains, hierarchical structures can be built for enterprises. In addition, an ID Domain can participate in multiple Correlating Domains, allowing peer structures to be built.

Peer structures offer significant potential for identification outside of enterprises. For example, any need to track a particular population could fund its own Correlating Domain and establish relationships with the appropriate ID Domains. The cost would be minimal for the Correlating Domain since it could take advantage of the accessibility to other Correlating and ID Domains. Furthermore, accessibility would be determined by the participating Domain. For example, each participating Domain would only allow access by authenticated, authorized requesters.

One additional point involves the likelihood of a National Healthcare ID. The PIDS model views the National Healthcare ID as its own ID Domain since assignment and control would not be under the control of local ID Domains. Thus this national ID would be a trait in ID Domains (and Correlating Domains) creating an automatic cross-reference for searches.

Person IdentificationService Module 2

Contents

This chapter contains the following topics.

2.1 Overview

Figure 2-1 on page 2-2 shows the interfaces defined in the **PersonIdService** module. The main functional interfaces inherit from the **IdentificationComponent** interface. They are also referenced by the **IdentificationComponent**.

PIDS Components and Inheritence Diagram

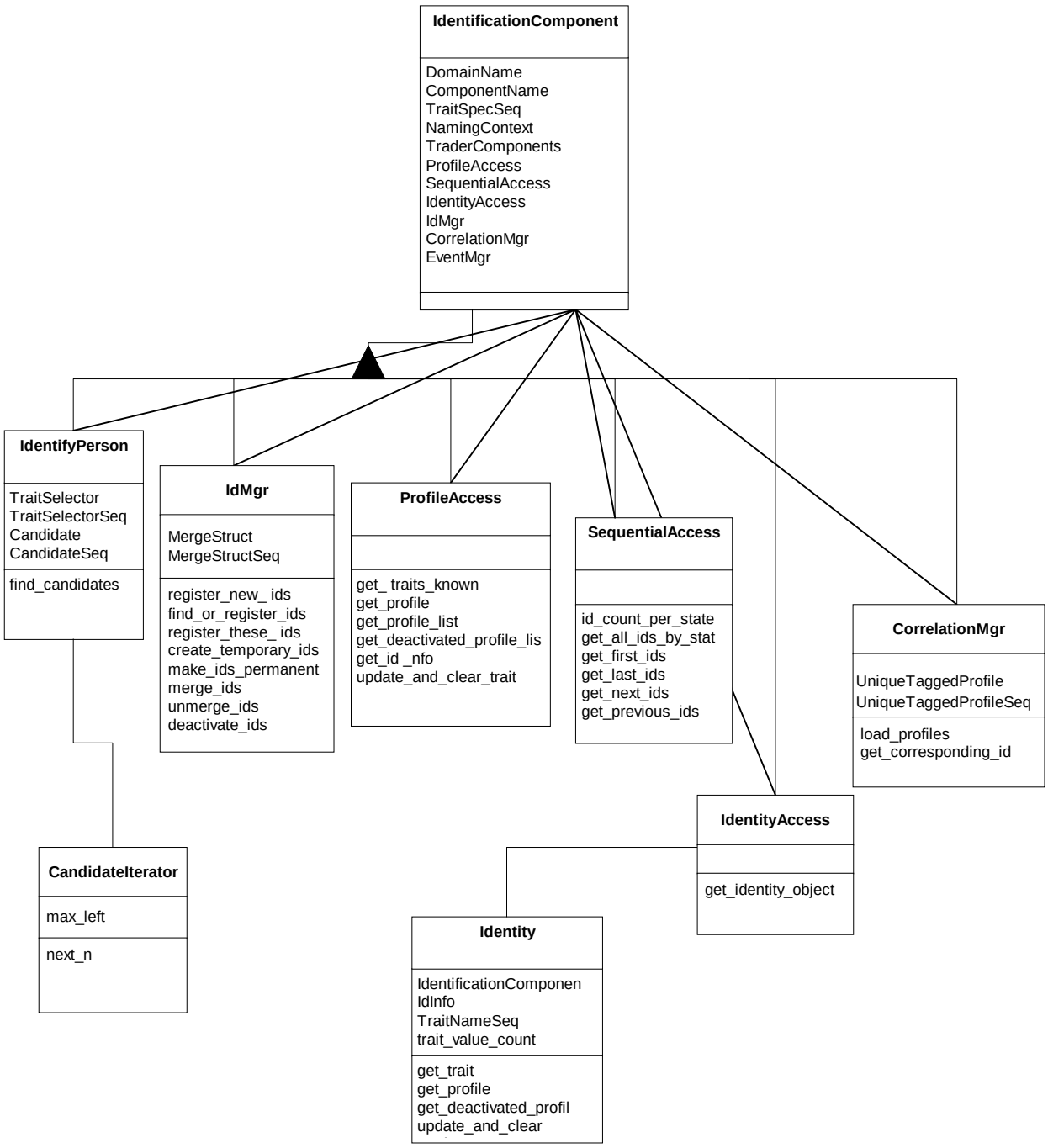


Figure 2-1 PIDS Components and Inheritance Diagram

```
//File: PersonIdService.idl

#ifndef _PERSON_ID_SERVICE_IDL_
#define _PERSON_ID_SERVICE_IDL_


#include <orb.idl>
#include <NamingAuthority.idl>
#include <Naming.idl>
#include <Trading.idl>
#include <Notification.idl>
#include <CosPropertyService.idl>

#pragma prefix "omg.org"

module PersonIdService
{
// . . .
};

#endif // _PERSON_ID_SERVICE_IDL_
```

The core Person Identification Service (PIDS) consists of the type declarations and interfaces in this module. PIDS is structured as a component that has multiple interfaces that may be implemented by any particular instance of the service. Each of the interfaces represent a particular piece of functionality and are optional. Even though an implementation of PIDS is free to implement which interfaces they need, specific interfaces are called out that must be implemented to claim conformance to the various conformance classes (see Chapter 6, Conformance Classes for additional information).

#include "NamingAuthority.idl"

The types declared within the **NamingAuthority** module are used for defining the names of ID Domains, IDs, components, and traits. The names of ID Domains, IDs, and components must be defined by implementations of the service or other parties. This specification defines some of the names for traits but they may be added independently by other parties. The global uniqueness properties of the **NamingAuthority** provides a mechanism for names to be created independently with low likelihood of name clashing.

#include "Naming.idl"

The **CosNaming** module is used for a couple of reasons. This specification defines some standard names for PIDS to use within a naming service. PIDS components also include a **NamingContext** that is used to provide federation.

#include "Trading.idl"

The **CosTrading** module is used by PIDS in various ways. This PIDS specification defines Trader Service types that PIDS components must use when offered to a service. This specification also uses many of the design patterns found in the Trader specification.

In addition PIDS components include a **TraderComponents** that is used to provide federation.

#include "Notification.idl"

The **Notification** module is used for defining change event notifications on the component.

#pragma prefix "omg.org"

To prevent name pollution and name clashing of IDL types, this module (and all modules defined in this specification) uses the pragma prefix that is the reverse of the OMG's DNS name.

2.2 Basic Types

```
// ----------------------------------------------------------------------------
// Common Data Types
//
typedef NamingAuthority::AuthorityId DomainName;
typedef sequence< DomainName > DomainNameSeq;

typedef NamingAuthority::LocalName PersonId;
typedef sequence< PersonId > PersonIdSeq;

struct QualifiedPersonId {
      DomainName domain;
      PersonId id;
};
typedef sequence< QualifiedPersonId > QualifiedPersonIdSeq;

typedef NamingAuthority::QualifiedNameStr TraitName;
typedef sequence< TraitName > TraitNameSeq;
typedef any TraitValue;
struct Trait {
      TraitName name;
      TraitValue value;
};
typedef sequence< Trait > TraitSeq;
typedef TraitSeq Profile;
typedef sequence< Profile > ProfileSeq;

enum IdState { UNKNOWN, INVALID, TEMPORARY, PERMANENT, DEACTIVATED };
typedef sequence<IdState> IdStateSeq;
struct IdInfo {
      PersonId id;
      IdState state;
      PersonId preferred_id;
};
typedef sequence<IdInfo> IdInfoSeq;

// ----------------------------------------------------
// Miscellaneous Data Types
```

```
//

typedef string ComponentVersion;
struct ComponentName {
      NamingAuthority::QualifiedName name;
      ComponentVersion version;
};

struct TraitSpec {
      TraitName trait;
      boolean mandatory;
      boolean read_only;
      boolean searchable;
};
typedef sequence< TraitSpec > TraitSpecSeq;

enum HowManyTraits { NO_TRAITS, SOME_TRAITS, ALL_TRAITS };
union SpecifiedTraits switch ( HowManyTraits )
{
      case SOME_TRAITS: TraitNameSeq traits;


struct TaggedProfile {
      PersonId id;
      PersonIdService::Profile profile;
};
typedef sequence<TaggedProfile> TaggedProfileSeq;

struct QualifiedTaggedProfile {
      QualifiedPersonId id;
      PersonIdService::Profile profile;
};
typedef sequence<QualifiedTaggedProfile> QualifiedTaggedProfileSeq;

struct ProfileUpdate {
      PersonId id;
      TraitNameSeq del_list;
      TraitSeq modify_list;
};
typedef sequence< ProfileUpdate > ProfileUpdateSeq;

struct MergeStruct {
      PersonId id;
      PersonId preferred_id;
};
typedef sequence< MergeStruct > MergeStructSeq;

struct TraitSelector {
      PersonIdService::Trait trait;
      float weight;
};
typedef sequence<TraitSelector> TraitSelectorSeq;

struct Candidate {
      PersonId id;
```

```
        float confidence;
        PersonIdService::Profile profile;
};
typedef sequence<Candidate> CandidateSeq;

interface CandidateIterator {
        unsigned long max_left();

        boolean next_n(
                in unsigned long n,
                out CandidateSeq ids );

        void destroy();
};

typedef unsigned long Index;
typedef sequence< Index > IndexSeq;

enum ExceptionReason {
        UNKNOWN_TRAITS,
        DUPLICATE_TRAITS,
        WRONG_TRAIT_FORMAT,
        REQUIRED_TRAITS,
        READONLY_TRAITS,
        CANNOT_REMOVE,
        MODIFY_OR_DELETE
};

struct MultipleFailure {
        Index the_index;
        ExceptionReason reason;
        TraitNameSeq traits;
};
typedef sequence< MultipleFailure > MultipleFailureSeq;

interface Identity;
typedef sequence< Identity > IdentitySeq;


// ------------------------------------------------------------
// Exceptions
//

exception InvalidId { IdInfo id_info; };
exception InvalidIds { IdInfoSeq id_info; };
exception DuplicateIds { PersonIdSeq ids; };
exception UnknownTraits { TraitNameSeq traits; };
exception DuplicateTraits { TraitNameSeq traits; };
exception WrongTraitFormat { TraitNameSeq traits; };
exception InvalidStates {};
exception TooMany { unsigned long estimated_max; };
exception MultipleTraits { MultipleFailureSeq failures; };

exception ReadOnlyTraits { TraitNameSeq traits; };
exception CannotRemove { TraitNameSeq traits; };
```

```
exception ModifyOrDelete { MultipleFailureSeq failures; };
exception NotImplemented {};

exception InvalidWeight {};
exception CannotSearchOn { TraitNameSeq traits; };

exception IdsExist { IndexSeq indices; };
exception RequiredTraits { TraitNameSeq traits; };
exception ProfilesExist { IndexSeq indices; };
exception DuplicateProfiles { IndexSeq indices; };
exception CannotMerge {IndexSeq indices; };

exception DomainsNotKnown { DomainNameSeq domain_names; };
exception IdsNotKnown { QualifiedPersonIdSeq ids; };
```

There are a number of structured types used widely throughout the **PersonIdService** module. These characterize ID Domains, IDs, traits of a person, etc.

2.2.1 Common Data Types

DomainName, DomainNameSeq

The **DomainName, DomainNameSeq** structure can be used to identify a fully qualified ID Domain name following the specified rules of the **NamingAuthority** module.

IDs for people are always relative to some ID Domain. An ID Domain is the extent of meaningfulness of the IDs within that ID Domain. The **DomainName** is an **AuthorityId** which makes it possible to define these independently without two ID Domains being named the same.

The **DomainNameSeq** is a sequence of **DomainNames**.

PersonId, PersonIdSeq

The **PersonId** is the simplest representation of an ID and is defined as a string. It can only be used in contexts where the ID Domain of the ID is either specified or is already known. The ID for a person relative to an ID Domain is represented as a name within that naming authority. A **PersonId** can only be interpreted in the context of an ID Domain.

A **PersonId** is the concrete representation of ID in an ID Domain. The representation of characters forming an ID is dependent on the specifics of the ID Domain. Given IDs from different ID Domains it is not possible to tell reliably what ID Domain they came from solely based on the syntax.

The **PersonIdSeq** is a sequence of **PersonIds**.

QualifiedPersonId, QualifiedPersonIdSeq

The **QualifiedPersonId** is a fully qualified ID since it contains the name of the ID Domain and the **PersonId** itself. If two **QualifiedPersonId**s are found to be identical, they can reasonably be assumed to represent the same person (within the limits of the **NamingEntity** algorithms of the **NamingAuthority** module if they are followed).

TraitName, TraitNameSeq

The names given to the traits of a person are characterized by the **TraitName** type. The trait name is a string in the form of the **QualifiedNameStr** so that additional traits can be created by users of the specification without having name clashes or having to register their trait names with a centralized authority.

The **TraitNameSeq** is a sequence of **TraitNames**. It is very useful for specifying the set of traits a client is interested in.

TraitValue

The **TraitValue** represents the value of a person's trait. Since the set of traits is virtually endless and could be of any type imaginable (including multimedia) the "any" data type is used.

Trait, TraitSeq, Profile, ProfileSeq

The characterization of a specific trait is given as a name value pair and is called "Trait." A sequence of these can be put together to create a **TraitSeq**.

A **Profile** is used when referring to the traits stored by a PIDS and bound to a **PersonId**. It is also used in the matching process for looking up persons. The term "Profile" is given to these sequences of traits since they are used a lot. At times a sequence of **Profiles** is needed (**ProfileSeq**).

IdState, IdStateSeq, IdInfo, IdInfoSeq

The **IdState** represents the set of states that an ID may take on. Each state has some well defined semantics. The **IdState** is a characteristic of an **ID**; whereas, traits are characteristics of the **person** represented by an ID. For this reason there is a separate operation to access it. The **IdState** can also be used for change event audit trails and change event notification. The operations on the **IdMgr** interface affect the state of an ID.

The clients to the interface see a different behavior for each ID state:

- **UNKNOWN** - This state indicates the service does not know if the ID exists or not and if so what the actual state might be. This is used by components that do not manage an ID Domain, but are a resident of an ID Domain where they may only know about a small subset of the IDs that exist.
- **INVALID** - An ID in this state can only be used in operations that create an ID. If a PIDS component knows all IDs in its ID Domain, every other possible ID has the Invalid **IdState**.
- **TEMPORARY** - An ID can be created as temporary without indicating any mandatory traits. A common usage is to create an ID that data can be bound to a patient before that patient is identified with an appropriate confidence. A temporary ID can be made permanent, merged, or deprecated. A temporary ID is made permanent explicitly - just updating the profile to contain all required traits does not change the state.
- **PERMANENT** - When an ID is created as permanent all mandatory traits must be provided. A permanent ID can be merged or deprecated but not made temporary.

- **DEACTIVATED** - Once an ID is expected not to be needed any more it can be deactivated (merged or deprecated), which keeps it around for historical purposes. A deactivated ID is in its final state and cannot be transitioned to any other state by PIDS operations, except unmerging.

The **IdStateSeq** is used for passing in the states of interest in a query. In a query, an **IdStateSeq** having a value of zero length should be interpreted to mean, return all valid states for the operation except **UNKNOWN** and **INVALID**.

The **IdInfo** structure contains the ID and the state of the ID. It also contains a preferred ID that is blank if in any state other than **DEACTIVATED**, or may contain the ID this ID has been superseded by (merged with) if the state is deactivated. The **IdInfoSeq** is a sequence of **IdInfo** structures.

ComponentVersion, ComponentName

The **ComponentVersion** is a numeric string with major and minor version indications. The part of the string up to the first period (".") is the major version indication, which must contain only numeric digits. The rest of the string after the first period is the minor version indication that may contain any printable characters. When a component is changed significantly enough that clients may need to react differently, the major version should be rolled. The minor version part of the string is used for vendor specific purposes. If only the minor version number of a component changes, clients can assume behavior from the service that is compatible with what came before. For example, a change to the ID Domain name, supported traits, or supported interfaces would require changing the major version number for a component since they are considered static over time.

The **ComponentName** includes a globally unique name and the version for a component. The name is used to uniquely identify instances of components. There is no inherent meaning implied by the component name.

2.2.2 Miscellaneous Data Types

TraitSpec, TraitSpecSeq

The **TraitSpec** characterizes a supported trait by listing its name and indicating if it can be changed, whether it is required for permanent IDs, and if it can be searched on.

- If the **read_only** field is true, then operations that attempt to modify it will throw an exception.
- If the mandatory field is true, then it can be assumed that any IDs in the Permanent ID state will have a value for that trait. This may be helpful in knowing what traits to ask for. In addition, if the **IdMgr** interface is implemented by the component, an exception may be raised if there is an attempt to create a permanent ID without a value supplied for the trait. Also an attempt to clear the value of the trait on a permanent ID raises an exception.
- If the searchable field is true, then this trait can be used for searches, assuming the component implements the **IdentifyPerson** interface. The **find_candidates()** operation will raise an exception for traits that have this field set to false.

A **TraitSpecSeq** is a sequence of **TraitSpec** structures and is used for specifying a complete set of traits supported by a component.

HowManyTraits, SpecifiedTraits

The **SpecifiedTraits** union is used by clients to indicate the traits they are interested in obtaining values for. The selector being **HowManyTraits** could indicate all of them or just some. If it indicates some, the union body has a list of the ones they are interested in.

TaggedProfile, TaggedProfileSeq

A **TaggedProfile** structure includes a profile and the ID associated with it. The **TaggedProfileSeq** is a sequence of **TaggedProfiles**.

QualifiedTaggedProfile, QualifiedTaggedProfileSeq

A **QualifiedTaggedProfile** contains the globally unique ID for a person and a profile for that ID. The **QualifiedTaggedProfileSeq** is a sequence of **QualifiedTaggedProfiles**.

ProfileUpdate, ProfileUpdateSeq

The **ProfileUpdateSeq** is used to update (modify) a list of profiles in a single call. Each entry (**ProfileUpdate**) contains the ID, the names of traits that need to be cleared, and the set of new trait values that need to be modified.

MergeStruct, MergeStructSeq

The **MergeStruct** is used to represent a request to deprecate one ID in preference to another one. The **MergedStructSeq** is used to batch a group of merges in a single call. The **preferred_id** field contains the ID that the **merged_id** is merged to.

TraitSelector, TraitSelectorSeq

The **TraitSelector** is the matching parameter used to tell the service identifying information about a person (or persons) that a client is interested in during a search. The structure contains a **Trait** plus the weight field as an indication of the relative weights to put on the trait. Since the matching engine is not being standardized by the PIDS, this weighting hint may be ignored by the service.

Index, IndexSeq

The **IndexSeq** is needed for some exceptions to return the indices of passed-in sequences that caused the exception.

ExceptionReason, MultipleFailure, MultipleFailuerSeq

The **ExceptionReason** enumeration is used to specify what the cause of the exception is in a **MultipleFailure** structure.

The **MultipleFailure** structure is used to report information back in an exception that can have problems with multiple traits each on multiple IDs or profiles passed in. The

"**the_index**" field contains the index from the original sequence passed to the operation that contained profiles or IDs. The "traits" field contains the names of the traits that had a problem at the index.

The **MultipleFailureSeq** is used to pass back problems that were found on multiple passed-in profiles or IDs.

IdentitySeq

The **IdentitySeq** contains a sequence of object references to the **Identity** interface. It is used for retrieving multiple **Identity** object references from the **IdentityAccess** interface with one call.

Candidate, CandidateSeq

The **Candidate** is returned after searching for persons when there may be partial matches. It contains the ID of the person, a confidence of how well that person's profile matches the profile selector, and the set of traits requested in the operation.

CandidateSeq is a sequence of candidates. These are returned from looking up a person in the service.

2.2.3 CandidateIterator Interface

CandidateIterator is an iterator for candidates. It is used when more IDs match a look up request than the client wants at one time.

max_left()

Returns the count of candidates left on the iterator.

next_n()

Returns the number of candidates asked for or all left on the iterator, whichever is less. Removes the returned candidates from the iterator before returning. If all candidates on the iterator are returned, the object is deleted upon returning the results.

destroy()

Destroys the iterator instance while there are still candidates on the iterator. Clients must always call this operation if they are finished accessing candidates from the iterator when more are left. It is a server implementation decision when to auto delete the iterator if the destroy is not called for a long time.

2.2.4 Exceptions

The following exceptions are generally useful by most or all of the interfaces of this module.

InvalidId

The InvalidId exception is generated when a single ID is passed into an operation but the ID is not valid for the operation. There are a number of reasons this may occur:

- The ID may not exist in the ID Domain that a service exists in.
- The operation being called may require the ID to be in a particular state.

The **IdInfo** for the ID is passed back to the caller so they may inspect the reason for the exception.

If the service knows of all IDs in an ID Domain (e.g., manages the IDs), then IDs that don't exist are returned as Invalid. This exception is not raised for UnknownIds. The difference in semantics to the client is that if they receive the exception with Unknown IDs they could try the manager of the ID Domain to see if the ID exists.

InvalidIds

The InvalidIds exception is similar to InvalidId but is used with operations that take more than one ID as part of the parameter(s). The complete set of violating IDs must be returned. A client could remove the violating IDs and repeat the operation knowing this exception will not be thrown again (unless the server state changed between calls).

DuplicateIds

The DuplicateIds exception is raised when the same ID is passed more than once in a sequence. The complete set of duplicate IDs are returned.

UnknownTraits

The UnknownTraits exception may be thrown for operations that take one or more trait names or traits as parameters (typically as sequences). The service throws the exception for traits it does not support. The complete set of violating traits must be returned. A client could remove the violating traits and repeat the operation knowing this exception will not be thrown again.

DuplicateTraits

The DuplicateTraits exception is raised when the same trait or trait name is passed more than once in a sequence. The complete set of duplicate trait names are returned.

WrongTraitFormat

The WrongTraitFormat exception may be thrown for operations that take one or more traits as parameters (typically as sequences). The service throws the exception if the **TraitValue** is of the wrong IDL type. The complete set of violating traits must be returned. A client could remove the violating traits and repeat the operation knowing this exception will not be thrown again.

InvalidStates

The InvalidStates exception is thrown when duplicate states are passed in, or the states

are passed in to operations that are not allowed.

TooMany

Many operations return a sequence of data that could be very large. If the size is larger than is feasible for the service to return (and the service can detect the fact), it will throw this exception. Typically the size of the returned result is determined partly by passed-in parameters. The service may return the estimated maximum size for a passed-in sequence. If that does not make sense for an operation or the service does not know what the maximum size would be, it can return zero. Alternatively, the service could return a size that it is confident will work even though it may not be close to the maximum. In either case, the client should be able to pass in a sequence of the **estimated_max** and this exception would not be thrown again.

Note the actual maximum size for a request may be a complex function based not only on the number of results requested, but also on which traits are requested, which traits have values on which IDs, the free memory on the server, etc. For this reason, the maximum size may not be known *a piori*.

MultipleTraits

The MultipleTraits exception is similar to the DuplicateTraits exception except that it applies to operations that could have problems with traits on multiple IDs or profiles that were passed in. Each entry in the sequence returned contains the index of the original profile or ID that had a problem and the trait names that had a problem at the index.

ReadOnlyTraits

The ReadOnlyTraits exception is thrown if a client tries to modify or set a trait that is read-only. The complete set of violating traits is returned to the client so they may remove them from the list and try again.

CannotRemove

The CannotRemove exception is raised when a client tries to clear a trait that is read-only or mandatory and the ID is in the Permanent **IdState**. The complete set of violating traits is returned to the client so they may remove them from the list and try again.

ModifyOrDelete

The ModifyOrDelete exception may be raised on operations that allow both clearing of traits and modifying/setting other traits. If a client passes in the same trait to be cleared and modified, this exception is raised.

NotImplemented

The NotImplemented exception is raised by the few optional operations on the interfaces. The operations that may raise this exception are logically grouped with other operations on the same interface. That is why these optional operations are not separate interfaces on **IdentificationComponent**.

InvalidWeight

The InvalidWeight exception is thrown if a weight is passed in for a trait selector that is less than 0.0 or greater than 1.0.

CannotSearchOn

The CannotSearchOn exception is raised if traits are passed in to be matched against when their searchable field is false.

IdsExist

If a client tries to create new IDs and they already exist, the IdsExist exception will be raised. The sequence indices for all violating IDs are returned.

RequiredTraits

If a client tries to create a permanent ID without giving all the mandatory traits, the service raises the RequiredTraits exception or returns a temporary ID instead.

Note that since a "trait" includes both a trait name and trait value, the Required Traits exception denotes the fact that the operation's inputs may have been lacking not only in "which traits had values," but in the *values themselves*. In both cases, the RequiredTraits exception would be thrown. However, in the former case the reason would be "RequiredTraits" and in the latter case the reason would be "InsufficientConfidence."

These specific reasons for failure are represented in the second element of the MultiplefailureSeq structure.

the_index	Identifies the entry that this structure responds to.
ExceptionReason	Indicates the specific reason for failure of this entry.
TraitNameSeq	If the input failed due to lack of some required traits, then this structure names the specific traits that were missing *for this entry.*
InsufficientConfidence	The specification of InsufficientConfidence as a failure reason in a make_ids_permanent operation indicates that the underlying logic has rejected the profile's confidence by virtue of trait *values*.

ProfilesExist

If a client tries to create new IDs and the profiles passed in correspond to IDs that exist, the ProfilesExist exception will be raised. The sequence indices for all violating profiles are returned.

Duplicate Profiles

If a client tries to create new IDs and two or more of the profiles passed in correspond to

each other, this exception will be raised. The sequence indices for all violating profiles are returned.

CannotMerge

This exception is raised when a client requests merging two IDs that the server knows cannot be merged. The indices returned correspond to each MergeStruct that could not be merged.

DomainsNotKnown

The DomainsNotKnown exception is thrown when an operation on the **CorrelationMgr** interface receives a parameter specifying an ID Domain name that is not part of the source domains it knows about. All violating **DomainNames** are returned.

IdsNotKnown

The IDs referenced are not known by the service. A complete list of violating IDs are returned.

2.3 *IdentificationComponent Interface*

```
// ----------------------------------------------------------------------------
// IdentificationComponent
//

interface ProfileAccess;
interface SequentialAccess;
interface IdentityAccess;
interface IdentifyPerson;
interface IdMgr;
interface CorrelationMgr;

interface IdentificationComponent
{
readonly attribute DomainName                   domain_name;
readonly attribute ComponentName                component_name;
readonly attribute TraitSpecSeq                 supported_traits;

readonly attribute IdentifyPerson               identify_person;
readonly attribute ProfileAccess                profile_access;
readonly attribute SequentialAccess             sequential_access;
readonly attribute IdentityAccess               identity_access;

readonly attribute IdMgr                        id_mgr;

readonly attribute CorrelationMgr               correlation_mgr;
readonly attribute Notification::EventComponent event_component;
readonly attribute CosNaming::NamingContext     naming_context;
```

```
readonly attribute CosTrading::TraderComponents    trader_components;
    void get_supported properties(
        in TraitName name,
        out CosPropertyService::Properties trait_defs)
            raises(
            UnknownTraits
};
};
```

The Person Identification Service is based on a component model similar in effect to the pattern used by the Trader Service. All identification interfaces are inherited from a core **IdentificationComponent**. The **IdentificationComponent** also has references to each of the other interfaces that are implemented. This makes it possible for a client to obtain an object reference to any of the identification interfaces and be able to easily find out what other functionality is implemented and to navigate to those interfaces.

The **IdentificationComponent** service encapsulates a logical table with person characteristics (traits) that is keyed by an ID, a matching engine that uses that table to map traits back to their ID, and a table of ID information (e.g., ID states).

This component can also be used by systems and applications (such as ancillary systems) that use IDs from an ID Domain but are not the manager of the IDs in the ID Domain. The IDs known by a system implementing this interface are likely to be a subset (possibly very small subset) of the IDs that exist for that ID Domain. Due to the fact that they may only be dealing with a few patient's data at any one time, the sequential access operations can be useful for systems with only a few IDs where a client application can request a page of data at a time and allow a user to scroll through the data.

Multiple systems in an ID Domain may implement the PIDS component interface and be binding identifying data (as well as other information beyond the scope of this specification) to IDs in that ID Domain. These may (or may not) also be exposed in the naming context and Trader Service of the ID Domain manager.

2.3.1 IdentificationComponent Interface

An **IdentificationComponent** has a number of optional interfaces that it may implement. There are a variety of systems and applications that may implement the interface for different reasons. They may pick and choose the functionality that is right for them. A component may be implemented by a single object (e.g., inheriting all the interfaces into one implementation-dependent interface), a different actual object for each interface supported (potentially distributed from each other), or any combination of the two. If multiple objects are used to implement the component, they must all maintain consistency so the client can treat them as one. That is, all the attributes on the component must return functionally identical results.

The interfaces implemented by a single component must operate over the same set of person IDs. Also the trait values must be consistent. Semantically they are using the same data base. If they are actually using separate data bases, it is up to the implementation to maintain consistency.

There are a wide variety of ways an **IdentificationComponent** can be used. The

following list demonstrates the wide variety of possible uses:

- ***Master Patient Index Systems*** - These systems correlate over multiple ID Domains and represent the fullest use of the various interfaces.
- ***Registration Systems*** - These may manage a single ID Domain and not know of other ID Domains. The registration, identification, and demographics services could be created as a single component even though they may be separate systems and possibly from separate vendors. Alternatively, the three separate services could be implemented as separate components.
- ***Ancillary Systems*** - Some ancillary systems may use IDs from the registration system and not manage an ID Domain at all. They would typically only know about a small population of the IDs in the ID Domain at any point in time. Examples of these could be Laboratory Information Systems (LIS), Radiology Information Systems (RIS), scheduling systems, monitoring systems, financial systems, pharmacy systems, etc. This represents the largest number of systems in a healthcare enterprise as they are doing specific functions that use the IDs as opposed to managing the IDs and ID information.

The first four interfaces (**ProfilesAccess**, **SequentialAccess**, **IdentityAccess**, **IdentifyPerson**) on this component are of general use and may be found on any type of component. They are the only interfaces that an ancillary system (that uses the medical record number) would typically implement (i.e., unless they maintained their own ID Domain). These basic interfaces give query access to lookup persons by matching against a passed-in profile, and query access to find the profile information about persons, given an ID.

There is, at most, one component per ID Domain that implements the **IdMgr** interface. Other components in the ID Domain use notification events from the ID Domain manager to keep their cached values of the traits they are interested in kept up to date.

A PIDS implementation may implement security restrictions that prevent access to IDs and/or certain traits for an ID by not returning information on it. This can make it appear the person is not known by the service or certain traits about the person are not known to the service. Alternatively, the service may raise security restriction exceptions.

domain_name

This is the name of the ID Domain the component resides in. Even stand-alone systems that implement their own ID Domain must create an ID Domain name and set this value. The entity responsible for naming the ID Domain must follow the rules defined in the **NamingAuthority** module for selecting names uniquely for each ID Domain created. The **DomainName** is invariant over time.

This is a globally unique, permanent name over the space of PIDS instances. It is expected that over time organizations will need to federate in ways they do not presently anticipate. It must never be necessary to modify a **DomainName** once it has been put into service.

component_name

Each implementation instance of the component must create a unique name for the component. The name chosen does not need any particular meaning. The unique names

make it possible for clients traversing a graph of components to recognize components they have encountered before. This way they can detect cycles in the graph and not visit a component multiple times when they don't need to. This part of the **component_name** is invariant over time. The version within the component name can change over time according to the rules set out by the **ComponentVersion** definition.

If there are two or more objects with the same component name they must be replicas of each other with identical functionality. The mechanism used to maintain consistency between the replicas is implementation-dependent and is not exposed as standard interfaces.

supported_traits

This indicates the set of traits supported by a component. Each trait is indicated whether it is mandatory, read-only, or searchable. There is no ordering of the traits assumed. The set of traits are static over time (see **ComponentVersion**).

naming_context, trader_components

These attributes are used to provide federation of PIDS components. Either one or both may be **NULL** as they are optional capabilities for any PIDS component. See Chapter 5 "Naming/Trader Interoperation" for more details.

identify_person

This interface provides a way to identify a person (find a potential ID) from the traits known about them.

profile_access

This is the main interface for accessing the traits associated with an ID.

sequential_access

This interface provides mechanisms for scrolling forward and backward through the set of IDs the component knows about. This interface is mostly useful for components that have a small number of IDs that can easily be scanned by a person.

identity_access

This interface provides access similar to **profile_access** but with the client first accessing a separate object per ID. This is needed to simplify certain security constraints by some implementations.

id_mgr

Only one component within an ID Domain implements this interface. If there are multiple actual systems implementing this interface, they should look like a single component. This is the interface implemented by a registration system. Other components within an ID Domain set this attribute to point to the ID manager of the ID Domain, if one exists and they have a reference to it.

correlation_mgr

This interface is only implemented by components that correlate over other ID Domains.

event_component

The event component gives the ability to connect components together such that changes on one component are communicated to the other. For example, IDs being created or deprecated and modifications of profiles.

get_supported_properties()

Descriptive information of a trait can be obtained by this method. For a given trait name one can get back a sequence of Property (Properties). The value of a Trait might be a coded string (with HL7 and vCard traits being supported examples). One of the properties could be the coding scheme used in the trait so that the value could be properly decoded on the client side. It also might be descriptive labels to be used by a client. It is recommended that the **PropertyName** returned by an identifiable coding scheme that could be looked up in the **TerminologyService** if desired.

PIDS clients and servers may compare trait definitions based on the combined property name and value. Trait definitions may also be mapped through and LQS/TQS as follows:

- The **Property.property_name** maps to the TQS **QualifiedCodeStr**, making the property itself unambiguous in its definition.
- The **Property.property.property_value** maps to the TQS local code.

The property names and values for the predefined traits (HL7 2.3 and vCard elements) are standardized in their respective modules of this specification.

2.4 IdentifyPerson Interface

```
// ------------------------------------------------------------------------
// IdentifyPerson
//

interface IdentifyPerson :
IdentificationComponent
{
void find_candidates(
        in TraitSelectorSeq profile_selector,
        in IdStateSeq states_of_interest,
        in float confidence_threshold,
        in unsigned long sequence_max,
        in unsigned long iterator_max,
        in SpecifiedTraits traits_requested,
        out CandidateSeq returned_sequence,
        out CandidateIterator returned_iterator )
    raises (
        TooMany,
        UnknownTraits,
```

```
            WrongTraitFormat,
            CannotSearchOn,
            DuplicateTraits,
            InvalidStates,
            InvalidWeight );
};
```

This service defines the functionality for querying an ID Domain or individual system with a specified set of trait values and their weights.

find_candidates()

Knowing some identifying information about a person (or group of people with common traits), a client can ask the service to find the candidate persons the service thinks may match those traits. The valid states that can be passed in by the client are Temporary, Permanent, and Deprecated.

The client can indicate the maximum number of candidates it wants passed back in the call using the **sequence_max** parameter. If the service matches more than that, an iterator is created containing the rest. It is the responsibility of the client to either retrieve all candidates from the iterator or to call the **destroy()** method. The client can also specify the maximum number that should be returned in the iterator via the **iterator_max** parameter.

Candidate iterators require transient objects that could have significant memory management impact on a server. If a server does not support candidate iterators, they return a NULL object reference for every invocation and clients can only obtain candidates that are returned as a sequence.

Using the **sequence_max** and **iterator_max** a client has the flexibility to do things like:

- get all candidates as a sequence,
- get all candidates via an iterator,
- get up to 30 candidates as a sequence and the rest up to 50 on an iterator,
- get up to 15 candidates as a sequence and all the rest via an iterator.

The candidates returned by the operation have a confidence indication where the larger the number the better the match. No candidate is returned that has a confidence indicator that is less than the confidence threshold. The interpretation of the confidence threshold value is consistent with that of the confidence indicator. This indicates how well the stored profile for that person matched the passed-in profile selector. The number of candidates returned could be zero if the matching engine does not find anything it feels matches close enough to return. Since there are so many algorithms this becomes a quality of service issue that is not standardized.

The range of values for the confidence indicator is 0.0-1.0 with 1.0 being the higher confidence (e.g., 100%). The exact semantics of the confidence indicator is determined by the service but some guidelines may help. The client should be able to compare two returned candidates and determine if they are of equal confidence or determine which is a higher confidence. If the returned candidates have different confidence values, then they

are returned in confidence order with the highest being returned first. If a service does not provide confidence determination, they may return 0.0 as the confidence value the same for all candidates. In this way, a client can know whether the service uses confidence values by whether the first candidate has a confidence of 0.0 or not.

The confidence value only has meaning relative to the single call it was returned from. There is no standard way to compare confidences returned from different services or from multiple calls to the same service.

Some matching engines may use discrete matching (such as the UNIX ‘grep’ utility) and others may use fuzzy matching (such as spell checkers or the use of phonic similarities). Even for discrete matching it is difficult to define exact semantics that could be applied to confidences since there are multiple traits and some traits contain multiple fields. Furthermore, the definitions would have to take into account how to combine the results from each field and the results from each trait to determine the confidence.

Matching engines map their matching capabilities to the defined confidence semantics as they see fit. How close they meet the expectations of the user is determined a Quality of Service (QoS) issue that is not standardized.

The ‘weight’ field is a hint from the client on how much preference it thinks the server should give to each profile passed in for determining matching confidences. It may be thought of as the confidence the client has in each trait of the profile selector.

The weight must be between 0.0 and 1.0. The weights are relative measures such that an exact match on a trait with weight = 0.5 results in twice the increase in confidence than an exact match on a trait with weight = 0.25.

A server implementation can ignore the weight field hint if it chooses. Using it is considered a Quality of Service (QOS) issue that is not standardized. The semantics here are just a way to define the measuring stick but does not require implementations to prove adherence. This is in part because fuzzy semantics cannot be measured.

The **traits_requested** parameter indicates the traits to be returned for every candidate if the candidate has values for that trait.

2.5 *ProfileAccess Interface*

```
// ------------------------------------------------------------------------------
// ProfileAccess
//

interface ProfileAccess :
IdentificationComponent
{
TraitNameSeq get_traits_known(
        in PersonId id  )
    raises (
        InvalidId );

Profile get_profile(
```

```
            in PersonId id,
            in SpecifiedTraits traits_requested )
        raises (
            InvalidId,
            UnknownTraits,
            DuplicateTraits );

    TaggedProfileSeq get_profile_list(
            in PersonIdSeq ids,
            in SpecifiedTraits traits_requested )
        raises (
            TooMany,
            InvalidIds,
            DuplicateIds,
            UnknownTraits,
            DuplicateTraits );

    TaggedProfileSeq get_deactivated_profile_list(
            in PersonIdSeq ids,
            in SpecifiedTraits traits_requested )
        raises (
            NotImplemented,
            InvalidIds,
            DuplicateIds,
            UnknownTraits,
            DuplicateTraits );

        void update_and_clear_traits(
            in ProfileUpdateSeq profile_update_spec )
        raises (
            InvalidIds,
            DuplicateIds,
            NotImplemented,
            MultipleTraits );

    IdInfoSeq get_id_info(
            in PersonIdSeq ids )
        raises (
            TooMany,
            DuplicateIds );
};
```

ProfilesAccess is the most basic interface used to get identity information for persons. This service provides the simplest set of functionality for any PIDS service.

get_traits_known()

This operation returns the set of **Traits** known about a person by the service.

get_profile(), get_profile_list()

The **get_profile()** operation returns the profile or subset of the profile that the component knows about the person. The passed-in traits indicate what subset of the profile that is being sought by the client.

The **get_profile_list()** is a shorthand mechanism for getting profiles for more than one ID at a time. This can be much more efficient than **get_profile()** for getting profiles on a lot of IDs since only one network round trip is required for all IDs as opposed to one per ID. The results are returned with exactly one value for each ID passed in. The results are tagged with the ID.

Both operations raise the InvalidId(s) exceptions if the IDs are not Permanent or Temporary.

Access rights to profiles are determined by the CORBAsec mechanisms (i.e., the user may not be able to access them).

get_deprecated_profile()

This is a special operation to get the profile of deprecated IDs. The ID passed in must be in the Deprecated state and known by the component or else the InvalidId exception is raised. A service may choose to not keep the old profiles around in which case they raise the NotImplemented exception. The results are returned with exactly one value for each ID passed in. The results are tagged with the ID.

update_and_clear_traits()

This operation is used to modify the profile of already existing IDs. The structures passed in specify which traits in the profile to be cleared, and which to change or add. Traits not mentioned to be cleared or changed remain the value they were before the call.

- If a PIDS component logs an audit trail of profile changes, this operation will cause an event to be logged.
- If a trait for an ID is listed to be cleared and changed, the ModifyOrDelete exception is raised.
- If the passed-in ID is not in the Temporary or Permanent IdState, the InvalidIds exception is raised.
- If all supported traits are read-only, the service raises the NotImplemented exception.

get_id_info()

This operation returns the current **IdStateInfo** for each ID passed in. The results are returned with exactly one value for each ID passed in. The results are tagged with the ID.

2.6 SequentialAccess Interface

```
// ------------------------------------------------------------------------------
// SequentialAccess
//

interface SequentialAccess :
    IdentificationComponent
{
    unsigned long id_count_per_state(
        in IdStateSeq states_of_interest )
    raises (
        InvalidStates );

    TaggedProfileSeq get_all_ids_by_state(
        in SpecifiedTraits traits_requested,
        in IdStateSeq states_of_interest )
        raises (
            TooMany,
            UnknownTraits,
            DuplicateTraits,
            InvalidStates );

    TaggedProfileSeq get_first_ids(
            in unsigned long how_many,
            in IdStateSeq states_of_interest,
            in SpecifiedTraits traits_requested )
        raises (
            TooMany,
            UnknownTraits,
            DuplicateTraits,
            InvalidStates );

    TaggedProfileSeq get_last_ids(
            in unsigned long how_many,
            in IdStateSeq states_of_interest,
            in SpecifiedTraits traits_requested )
        raises (
            TooMany,
            UnknownTraits,
            DuplicateTraits,
            InvalidStates );

    TaggedProfileSeq get_next_ids(
            in PersonId reference_id,
            in unsigned long how_many,
            in IdStateSeq states_of_interest,
            in SpecifiedTraits traits_requested )
        raises (
            TooMany,
```

```
            InvalidId,
            UnknownTraits,
            DuplicateTraits,
            InvalidStates );

    TaggedProfileSeq get_previous_ids(
            in PersonId reference_id,
            in unsigned long how_many,
            in IdStateSeq states_of_interest,
            in SpecifiedTraits traits_requested )
        raises (
            TooMany,
            InvalidId,
            UnknownTraits,
            DuplicateTraits,
            InvalidStates );
};
```

id_count_per_state()

This operation indicates the number of IDs, having one of the ID states passed in, that the component knows of. The value could be zero at any point in time or it could be very large. Hospital ancillary systems would likely have from less than a dozen up to a hundred or many thousands. The valid states that can be passed in are Temporary, Permanent, and Deprecated.

get_all_ids_by_state()

This operation returns profiles for all patients the service knows about that match one of the states passed in. The returned profiles only contain the traits indicated by the passed-in parameter, if they exist for the ID. The valid states that can be passed in are Temporary, Permanent, and Deprecated.

get_first_ids(), get_last_ids(), get_next_ids(), get_previous_ids()

These are the operations that provide sequential access to all the IDs known by the system/service. The service must have a consistent way to order the IDs. The ordering may be different for each set of traits asked for; however, for a particular set of traits, the ordering is the same for all these operations. The ordering mechanism is implementation-dependent and hidden behind the interface. The valid states that can be passed in are Temporary, Permanent, and Deprecated.

A client can request the profiles for a number of persons at the beginning and end of the ordered list by using the **get_first_ids()** and **get_last_ids()** respectively. The client passes in the number of IDs and profiles wanted.

A client can also request a number of profiles that are either after or before a particular ID via the **get_next_ids()** and **get_previous_ids()**. The ID and profile for the ID passed in is not returned.

Using these four operations: **get_first_ids()**, **get_last_ids()**, **get_next_ids()**, and **get_previous_ids()** a client can scroll forward or backward through the set of IDs,

known by the service, a page at a time. The number of profiles returned may be smaller than that requested if the number of profiles held by the service is smaller than that requested.

2.7 IdentityAccess Interface

```
// ------------------------------------------------------------------------------
// IdentityAccess
//

interface IdentityAccess :
    IdentificationComponent
{
    Identity get_identity_object(
            in PersonId id )
        raises (
            InvalidId );

    IdentitySeq get_identity_objects(
            in PersonIdSeq ids )
        raises (
            InvalidIds );
};

interface Identity
{
    readonly attribute IdentificationComponent source_component;
    readonly attribute IdInfo id_info;
    readonly attribute TraitNameSeq traits_with_values;
    readonly attribute long trait_value_count;

    Trait get_trait(
            in TraitName trait_requested )
        raises (
            UnknownTraits );

    Profile get_profile(
            in SpecifiedTraits traits_requested )
        raises (
            UnknownTraits,
            DuplicateTraits );

    Profile get_deactivated_profile(
            in SpecifiedTraits traits_requested )
        raises (
            NotImplemented,
            UnknownTraits,
            DuplicateTraits );

    void update_and_clear_traits(
```

```
            in ProfileUpdate profile_update_spec )
        raises (
            NotImplemented,
            UnknownTraits,
            WrongTraitFormat,
            ModifyOrDelete,
            ReadOnlyTraits,
            CannotRemove,
            DuplicateTraits );

    void done();
};
```

The **Identity** object can provide individual ID level access control via CORBA Security. Specific access control parameters might be available in some security policies applied to an **Identity** object with the specific object being security-unaware. These policies can be set up to prevent access on an operation/attribute basis. The service must be security-aware in order to implement security policies that vary by the parameters passed in (e.g., separate access control for each trait).

Policies governing access to the interfaces are determined by administrative controls beyond the scope of this specification.

get_identity_object()

Returns an identity object that represents the ID passed in, assuming it is a valid ID and known by the component.

get_identity_objects()

Returns an identity object for each ID passed in, assuming they are valid IDs and known by the component.

2.7.1 Identity Interface

The **Identity** interfaces provide a way that access control can be applied at the ID. Instances of Identities are accessed from the **IdentityAccess** interface.

source_component

This is an object reference back to the component that created the **Identity** object. This may be useful if the reference was passed to a third party that may need to get to other functionality of the component.

id_info

The **id_info** attribute contains the ID by which this person is known. This consists of the **PersonId** (simple name) and the **IdState** of the ID. If it has been merged, it has the preferred ID as well.

traits_with_values

This read-only attribute returns the list of trait names for traits with values for the ID.

trait_value_count

This attribute is the count of the number of traits with values for the ID.

get_trait()

This operation returns the value set on the specified trait or raises an exception if the trait is not known.

get_profile()

This operation returns the traits requested that have values set.

get_deactivated_profile()

This is a special operation to get the profile of deprecated IDs. The ID must be in the Deprecated state or the InvalidId exception is raised. A service may choose to not keep the old profiles around in which case they raise the NotImplemented exception.

update_and_clear_traits()

This operation updates the profile by clearing (deleting the values) for some traits and setting the values for others. The ones not mentioned are left the same as before the call. If all supported traits are read-only, the service raises the NotImplemented exception.

done()

This is called when the client no longer needs to access the object. Since there could be many **Identity** objects created for a single PIDS component the proper use of this operation can help the service in its own memory management. The server may delete the object reference if the object is transient or may delete the object from memory for persistent objects. This does not preclude **Identity** objects from being implemented persistently but it does not guarantee they are persistent. It is a service implementation issue as to when transient **Identity** objects are deleted if **done()** is not called. Clients should be prepared to get another object reference from the **IdentityAccess** interface if the **Identity** object is no longer valid while they still need to use it.

2.8 IdMgr Interface

```
// ------------------------------------------------------------------------------
// IdMgr
//

interface IdMgr :
    IdentificationComponent
{
    PersonIdSeq register_new_ids(
```

```
            in ProfileSeq profiles_to_register )
        raises (
            ProfilesExist,
            DuplicateProfiles,
            MultipleTraits );

    PersonIdSeq find_or_register_ids(
            in ProfileSeq profiles_to_register )
        raises (
            DuplicateProfiles,
            MultipleTraits );

    void register_these_ids(
            in TaggedProfileSeq profiles_to_register )
        raises (
            NotImplemented,
            IdsExist,
            DuplicateIds,
            ProfilesExist,
            DuplicateProfiles,
            MultipleTraits );

    PersonIdSeq create_temporary_ids(
            in ProfileSeq profiles_to_register )
        raises (
            MultipleTraits );

    PersonIdSeq make_ids_permanent(
            in PersonIdSeq ids_to_modify )
        raises (
            InvalidIds,
            DuplicateIds,
            RequiredTraits );

    IdInfoSeq merge_ids(
            in MergeStructSeq ids_to_merge )
        raises (
            InvalidIds,
            DuplicateIds );

    IdInfoSeq unmerge_ids(
            in PersonIdSeq ids_to_unmerge )
        raises (
            InvalidIds,
            DuplicateIds );

    IdInfoSeq deprecate_ids(
            in PersonIdSeq ids_to_deprecate )
        raises (
```

```
                    InvalidIds,
                    DuplicateIds );
};
```

IdMgr is an interface providing the core set of functionality for managing IDs in a single ID Domain. Specifically it can allocate a unique ID to a specific profile. All the operations on **IdMgr** are “write” commands as opposed to “read” commands. For this reason, there may be more secure access control to prevent unwanted changes. The access control is managed by the CORBA security service and the implementation of the **IdMgr** service.

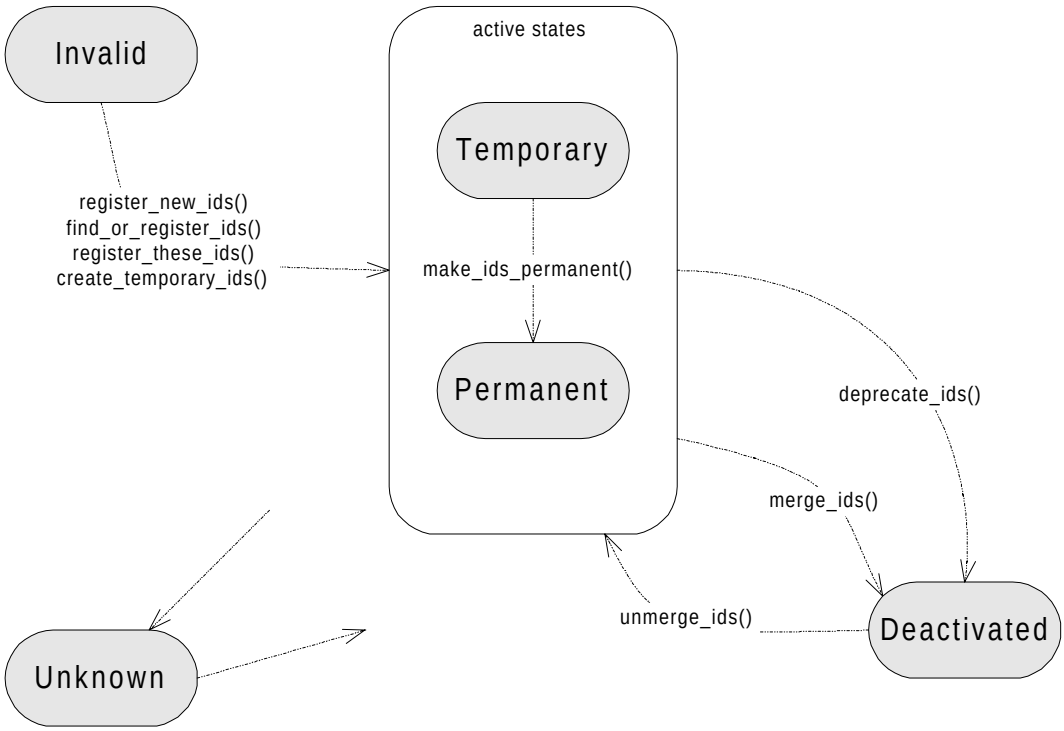


Figure 2-2 Transitions between IDStates

Figure 2-2 shows the transitions between **IdStates** that are defined by operations on the **IdMgr** interface. Using the **IdState** as the values for a state machine pulls these operations together into a coherent set of semantics. The transitions shown are only suggestive as the policies of the service are implementation-dependent. If a client calls a particular operation, they are suggesting to the service that certain transitions should occur. The service may make the transition immediately, may wait for human review before making the transition, or possibly never make the transition.

When a client needs to create a new ID, the service returns it but does not guarantee whether it is Temporary or Permanent. Services are not required to use the Temporary ID state.

register_new_ids()

This generates new IDs in this ID Domain and binds the passed-in profiles to them. The

register_new_ids() operation establishes an association between IDs and profiles making them available for other component operations. This also Creates distinct, new IDs according to the rules and conventions for the ID Domain. These rules and conventions are implicit to the ID Domain and are not accessible via standard CORBA IDL.

By calling this operation, the client is indicating it needs the IDs generated immediately and they are expected to become permanent. It is up to the service whether they are returned as permanent or as temporary IDs and require human interaction before becoming permanent.

find_or_register_ids()

This operation is used for generating IDs in unattended operation. An internal (hidden) value may indicate the confidence threshold that matching must attain in order to consider the person already has an ID in this Domain. Below this threshold a new ID is generated and the profile is bound to it.

This operation could produce two different resulting **IdStates**. It only changes the state of an ID if it creates a new ID. The resulting state of a newly created ID may be Temporary or Permanent which is implementation- and site-dependent. Some sites may choose to only produce temporary IDs automatically so they may be verified by a person before making them permanent.

register_these_ids()

This operation works similar to **register_new_ids()** except the client also indicates the value for the created IDs. The policy for generating the ID values may be hidden within the service and it would not allow clients to set the ID values. In this case, the service raises the NotImplemented exception.

create_temporary_ids()

The **create_temporary_ids()** operation creates new IDs and indicates the client needs IDs that may be Temporary. A PIDS implementation may return a Permanent ID or Temporary ID. The mandatory traits are not required to create a temporary ID. This operation will still create a new ID even if an existing profile matches the one passed in.

make_ids_permanent()

Temporary IDs may be made permanent by the **make_ids_permanent()** operation. For some services the permanent IDs returned may be the same as the one passed in. Other services may want to use a separate part of the ID name space for temporary IDs so they can tell the IDs are temporary by looking at them. For example, a service may precede every temporary ID by the letters TEMP-.

merge_ids()

If a person is found to have more than one ID in an ID Domain, all but one of them can be merged into the preferred ID by calling the **merge_ids()** operation with each duplicate ID. The preferred ID will be unchanged except if it supports the **PersonIdTraits::MergedIds** trait, it will be modified to reference the ID being merged

to it. The merged ID will have its **IdState** set to Deactivated (possibly after human review) and have the **preferred_id** field on its **IdInfo** set to the preferred ID. The preferred ID will have its **MergedIds** trait set with the merged ID in it.

The **IdInfoSeq** returned has the **IdInfo** for each of the target IDs passed in to the operation.

unmerge_ids()

If an ID that has been merged with another is found later to represent a different person, it may be unmerged with this operation. If the **MergedIds** trait is supported, the profile for the ID this was merged with is changed to no longer reference this ID.

If the passed-in ID(s) is(are) not merged (in the Deactivated state and **preferred_id** set to something other than (“”), the InvalidIds exception is raised.

The **IdInfoSeq** returned has the **IdInfo** for each of the target IDs passed in to the operation.

deprecate_ids()

Once an ID is expected to never be used again it may be retired from service by the **deprecate_ids()** operation which is a request to change the **IdState** to Deactivated. A service may either remove the profile for the ID or leave it intact for historical purposes, but it can never be changed. The profile can be accessed only with the special operation **get_deprecated_profiles()**.

The **IdInfoSeq** returned has the **IdInfo** for each of the target IDs passed in to the operation.

2.9 CorrelationMgr Interface

```
// ------------------------------------------------------------------------------
// CorrelationMgr
//

interface CorrelationMgr :
    IdentificationComponent
{
    readonly attribute DomainNameSeq source_domains;

    void load_profiles(
            in QualifiedTaggedProfileSeq tagged_profiles )
        raises (
            UnknownTraits,
            WrongTraitFormat,
            DomainsNotKnown );

    QualifiedPersonIdSeq get_corresponding_ids(
            in QualifiedPersonId from_id,
            in DomainNameSeq to_domains )
```

```
        raises (
            DomainsNotKnown,
            IdsNotKnown );
};
```

The **PersonIdTraits::CorrelatedIds** trait is a special trait that is only used by components that implement the **CorrelationMgr** interface. The **CorrelationMgr** is responsible for setting this trait to reference all source IDs that are correlated to the same ID in the correlation Domain. This must be consistent with the information obtained via the **get_corresponding_ids()** operation.

The **CorrelationMgr** interface does not provide operations to support manual correlation or retrospective verification of unattended correlation. These capabilities may be addressed in a later RFP.

source_domains

This read-only attribute contains a list of source ID Domains that profiles may be loaded from and IDs may be mapped to or from.

load_profiles()

This operation causes the profiles to be loaded into the Correlating ID Domain, from the specified source ID Domains. It is an implementation decision whether correlation is performed immediately.

Note – The use of the UnknownTraits and WrongTraitFormat exceptions has been deprecated for servers to raise. They remain on the interface so that new clients can still receive the old exceptions from old servers.

---

get_corresponding_ids()

This operation returns the IDs in the destination ID Domains that correspond to the ID passed in.

find_or_register_ids()

If this operation is implemented, it causes the profiles to be loaded into the Correlating ID Domain, from the specified source domains. IDs from the Correlating ID domain for each profile are returned.

NamingAuthority Module 3

Contents

This chapter contains the following topics.

3.1 NamingAuthority IDL

```
//File: NamingAuthority.idl

#ifndef _NAMING_AUTHORITY_IDL_
#define _NAMING_AUTHORITY_IDL_

#include <orb.idl>

#pragma prefix "omg.org"

module NamingAuthority
{
    enum RegistrationAuthority {
        OTHER,
        ISO,
        DNS,
        IDL,
        DCE };
```

```
    typedef string NamingEntity;

    struct AuthorityId {
        RegistrationAuthority  authority;
        NamingEntity           naming_entity;
    };
    typedef string AuthorityIdStr;

    typedef string LocalName;
    struct QualifiedName {
        AuthorityId authority_id;
        LocalName local_name;
    };
    typedef string QualifiedNameStr;

    exception InvalidInput {};

    interface translation_library
    {
        AuthorityIdStr authority_to_str(
                in AuthorityId authority )
            raises(
                InvalidInput );

        AuthorityId str_to_authority(
                in AuthorityIdStr authority_str )
            raises(
                InvalidInput );

        QualifiedNameStr qualified_name_to_str(
                in QualifiedName qualified_name )
            raises(
                InvalidInput );

        QualifiedName str_to_qualified_name(
                in QualifiedNameStr qualified_name_str )
            raises(
                InvalidInput );
    };
};

#endif // _NAMING_AUTHORITY_IDL_
```

The **NamingAuthority** module provides a means of giving globally unique names to name spaces and hence the names within those name spaces. The fundamental need is the ability to compare two names for equality. If they are equal, they are known to represent the same entity, concept, or thing. This is needed when independent entities are generating names that may get compared for equality; however, the reverse is not guaranteed to be true (that is, an entity may have several names).

The authority for the name space may derive from several different types of roots, the choice of which depends upon the user requirements as each root has different qualities of management and uniqueness. The various root types are defined below.

#pragma prefix "omg.org"

To prevent name pollution and name clashing of IDL types this module (and all modules defined in this specification) uses the pragma prefix that is the reverse of the OMG's DNS name.

RegistrationAuthority

Identifies the root of the name space authority. An entity (e.g., person or organization) may be registered with multiple different roots (RegistrationAuthorities) and be able to assign names and other name spaces within each root. These may be used for the same or for different needs. For this reason, there is no guarantee of any equality in the different name spaces managed by an entity. There are currently no means available to determine whether a given authority in an ISO hierarchy is the same authority as one specified in a DNS hierarchy.

- **OTHER**: This form of a naming authority should be used sparingly, and only in experimental or localized situations or special purposes. It is the responsibility of the implementing institution to guarantee uniqueness within the names themselves, and there is no uniqueness guarantee outside of the source institution. Services that define default naming authorities (and possibly also names) may also use the Other root to forego long **AuthorityId**s. In this case, the specification of the service must name **AuthorityId**s that may be expected with the Other root and still maintain name space integrity for that service.
- **ISO** (International Standards Organization[1]): The ISO specifies a registration hierarchy identified by a series of named/numbered nodes. Many of the coding schemes used in the medical environment are or can be registered within the ISO naming tree. The ISO root form is one of the recommended forms when the naming authority is internationally recognized, such as international coding schemes, or when the authority is to be used across two or more different enterprises. ISO provides for the recording of a responsible person and address for each node in the authority hierarchy.

1. ISO/IEC 8824-1 (1994) Information Technology - Abstract Syntax Notation One (ASN.1) - Specification of Basic Notation.

- **DNS**: Domain Name Services[2]. Internet domains are recorded with a central, global registration authority. Subhierarchies within the domains are then maintained locally by the registered organization or person. The DNS form is recommended as an alternative to the ISO naming tree when the specific naming authority needs identity and uniqueness, but is not in an ISO registration. By using this common characteristic of many organizations it gives the ability to create globally unique name spaces and names without the need to register as an ISO name authority. It is up to the organization itself to maintain the integrity of the name space(s) (e.g., not reusing names or name spaces).
- **IDL**: The OMG Interface Repository[3]. The CORBA Architecture specifies a means of identifying entities as being unique within the interface repository, via the use of a *RepositoryId*. CORBA repository ids may be in either the OMG IDL format, the DCE UUID format, or the LOCAL format. Within this specification, the "IDL" root refers only to the IDL format. The DCE format may be represented within the DCE root and the Local format within the Other root. The IDL authority may prove very useful when registering CORBA/IDL specific objects such as value sets, interface specifications, etc. It should be noted that OMG does not currently manage the repository name space in any rigorous fashion, and it is quite possible that two different developers may arrive at exactly the same repository ID for entirely different entities. For this reason, some people give the repository ID a prefix that consists of their reverse DNS that is "/" separated instead of "." separated. This root type may be very useful when the names within the name space are defined in IDL. For example, it could be the **RepositoryId** for an enumerated type or a module that has constant integers or strings defined for each name within the name space.
- **DCE**: The Distributed Computing Environment[4]. While they don't actually register coding schemes or other entities, they do provide a means of generating a globally unique 128-bit ID, called a Universally Unique ID (UUID). This UUID may be used to guarantee the uniqueness of a name space in situations where it is not necessary for the identity of the authority to be known outside of the specific implementation.

NamingEntity

Identifies a specific name in the syntax and format specified by the corresponding registration authority. The various naming authorities tend to provide a fair amount of leeway as far as the actual format of the registered names. As there may be situations where the full semantics of a specific authority's name comparison will not be available to an application, we have chosen to select a specific subset of the syntax of each representation. The intention is to be able to determine whether two registered entities are identical, or not, solely through the use of string comparison. The specific name formats

2. P. Mockapetris, " Domain Names - Concepts and Facilities", RFC 1034, Information Sciences Institute, November 1987.

3. OMG's *The Common Object Request Broker: Architecture and Specification*.

4. DCE 1.1 : Remote Procedure Call. OpenGroup Document Number C706, August 1997.

are described below:

- **OTHER**: An arbitrary string, syntax undefined except locally by a specific service specification and/or by particular implementations and installations. The "/" character is illegal to use as it is reserved as a separator of components in the stringified version of QualifiedName.
- **ISO**: The name should be represented using the *NameForm* of the *ObjectIdentifierValue* as specified in ISO/IEC Recommendation 8824-1. Each name component should be separated by a single space.

  Example: "joint-iso-ccitt specification characterString"
- **DNS**: The domain name and path in the form mandated in RFC 1034. The path name is represented as a dot separated tree which traverses up the hierarchy. Since DNS names are not case-sensitive, only lower case letters should be used such that simple string comparisons can determine equality. However, it is permissible to use case insensitive comparisons as well.

  Example: "pidsserv.slc.mmm.com"
- **IDL**: The OMG **RepositoryId** format specified in *The Common Object Request Broker: Architecture and Specification*, in the form: "<node>/<node>/…/<node>." The IDL: prefix and the version number suffix should NOT be used for the **NamingEntity**. The IDL: prefix is prepended to create the **AuthorityIdStr**.

  Example:"CosNaming/NamingContext/NotFoundReason" is the NamingEntity for:

```
module CosNaming {
    ...
    interface NamingContext {
        ...
        enum NotFoundReason { ... };
        ...
    };
};
```

- **DCE**: The UUID in the external form <nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn>, where <n> represents one of the digits 0-9 and the characters A-F. The alpha characters should all be upper case.

  Example: "6132A880-9A34-1182-A20A-AF30CF7A0000"

AuthorityId, AuthorityIdStr

The combination of a Registration Authority and Naming Entity, which identifies a specific naming authority. In situations where a given naming entity may have more than one naming authority, it should be agreed upon in advance which of the specific names for the entity is to be used. This specification makes no guarantees about the ability to recognize, for example, that an authority in the ISO structure is identical to an authority within the IDL structure.

The string version (**AuthorityIdStr**) is useful for situations where unique names are required in a string format. The string is created as <stringified **RegistrationAuthority**>:<**NamingEntity**>.

LocalName, QualifiedName, QualifiedNameStr

A local name is a name within (relative to) a namespace. It is simply a string representation.

A **QualifiedName** is a globally unique name for an entity by the fact that it carries the naming **AuthorityId** of the name space and the **LocalName** within that name space.

The **QualifiedNameStr** is a stringified version of the **QualifiedName**. The format of the string is **<stringified RegistrationAuthority**>: <**NamingEntity**>/<**LocalName**>. Notice that even though the slash character "/" cannot be used within the name of a **NamingEntity** it can be used within the **LocalName**. The following table summarizes the format for **QualifiedNameStr**. Columns 1-3 give the format for an **AuthorityIdStr**.

Registration Authority	(1) Stringified Registration Authority	(2) RA-NE Delimiter	(3) NamingEntity Format	(4) NE-LN Delimiter	(5) LocalName Format
OTHER	""	":" optional	<no '/'>	"/" optional	<no '/'>
ISO	"ISO"	":"	<use ISO rules>	"/"	<any characters>
DNS	"DNS"	":"	<use DNS rules>	"/"	<any characters>
IDL	"IDL"	":"	<use IDL rules>	"/"	<no '/'>
DCE	"DCE"	":"	<use DCE rules>	"/"	<any characters>

The definitions for type OTHER are defined to allow using a **QualifiedNameStr** format in contexts where an IDL "string" is currently used. A normal IDL string is a **QualifiedNameStr** with no **RegistrationAuthority** and no **NamingEntity**. The limitation is that any normal IDL strings that start with one of the **RegistrationAuthority** strings cannot be mapped into the **QualifiedNameStr** since they would be interpreted by the rules in this module.

The string for the "OTHER" type of **RegistrationAuthority** being a blank string ("") makes it easier for locally defined names to be usable with no requirements on the format except they cannot start with one of the strings reserved for the other **RegistrationAuthority** types. The ":" delimiter is optional for type OTHER. If the **NamingEntity** is "" for type OTHER, then the "/" delimiter is also optional.

3.2 Exceptions

InvalidInput

The InvalidInput exception is raised when the input parameter for the **TranslationLibrary** interface operations is determined to be of an invalid format.

3.3 TranslationLibrary Interface

This interface is meant to be a local library for translating between the structured version and stringified version of **AuthorityIds** and **QualifiedNames**.

authority_to_str,
str_to_authority,
qualified_name_to_str,
str_to_qualified_name()

Each of these operations take either a structured version or stringified version of a type and return the opposite. The data content returned is the same as that passed in. Only the representation of the data is changed.

Traits 4

Contents

This chapter contains the following topics.

The definitions in the **PersonIdService** module define traits as name/value pairs with the name being a string (in the **QualifiedNameStr** format) and the value being an “any.” This is done to provide complete flexibility in defining traits. The traits used to help identify a person may be of many types and formats. Many of them are represented as strings but others can be structured types and even multimedia types. In addition there are many inconsistent standards even for the more common traits like a person’s name.

This specification was written to allow the use of different traits and different formats of traits to be used as needed. A specific implementation of PIDS may use the traits that meet these needs. The formats of even the common traits may need to be different depending on the technical and political environment in which they are deployed. They may need to work with other information services that have a predefined data format that the implementation decides to use. The data formats used for traits may be required by legal means as well.

While this specification does not limit what traits can be used with PIDS it does specify a set that can be well known and used by clients and servers if they choose. The set consists of data formats from a couple of well-known industry standards as well as some special traits defined just for PIDS. The well-known standards are the Health Level Seven (HL7) and vCard™.

If there are environments where other standard sets of traits are needed, they could be standardized via the OMG's processes. These could be areas that require interoperable solutions to existing standards or legacy systems for which PIDS interfaces need to be provided.

Within the IDL **RegistrationAuthority**, **LocalName** can have any characters. Also, the **IdentificationComponent** has a string, **QualifiedNameStr component_name**, to name the particular PIDS service. In order to create and manage unique IDs, it is acceptable to include the **component_name** in the **LocalName** of the **Qualified PersonId** to further delimit the name space.

Thus, **QualifiedPersonIds** could have information in them, which allows one

- to determine the specifics of the server (accessible through the NamingService, for example).
- to locate the id of the person, to record types of identifiers, or even to qualify the identifier domain itself as a namesapce for its Ids.

The following sections define the three modules that contain the default set of traits defined by PIDS:

- **PersonIdTraits** Module
- **HL7Version 2_3** Module
- **vCardTraits** Module

4.1 PersonIdTraits Module

```
//File: PersonIdTraits.idl

#ifndef _PERSON_ID_TRAITS_IDL_
#define _PERSON_ID_TRAITS_IDL_

#include <orb.idl>
#include <PersonIdService.idl>

#pragma prefix "omg.org"

module PersonIdTraits
{
    const PersonIdService::TraitName NULL_TRAIT = "";
    typedef any NullTraitType;  // set to tk_null

    const PersonIdService::TraitName INTERNAL_ID = "PIDS/InternalId";
    typedef PersonIdService::PersonId InternalIdType;

    const PersonIdService::TraitName MERGED_IDS = "PIDS/MergedIds";
    typedef PersonIdService::PersonIdSeq MergedIdsType;

    const PersonIdService::TraitName DUPLICATE_IDS = "PIDS/DuplicateIds";
    typedef PersonIdService::PersonIdSeq DuplicateIdsType;
    const PersonIdService::TraitName CORRELATED_IDS = "PIDS/CorrelatedIds";
    typedef PersonIdService::QualifiedPersonIdSeq CorrelatedIdsType;
```

```
        const PersonIdService::TraitName EXTERNAL_IDS = "PIDS/ExternalIds";
        typedef PersonIdService::QualifiedPersonIdSeq ExternalIdsType;

        const PersonIdService::TraitName NATIONAL_HEALTH_IDS = "PIDS/NationalHealthId";
        struct NationalHealthIdStruct {
            string country_code;
            PersonIdService::PersonId id;
        };
        typedef sequence< NationalHealthIdStruct > NationalHealthIdStructSeq;
        typedef NationalHealthIdStructSeq NationalHealthIdsType;
        };

        typedef sequence<QualifiedPersonIdInfo> QualifiedPersonInfoSeq;
        const PersonIdService::TraitName EXTERNAL_CODED_IDS = “PIDS/ExternalCodedIds”;
        typedef PersonIdService::QualifiedPersonIdInfoSeq ExternalCodedIdsType;

        typedef NamingAuthority::AuthorityId CodingSchemeId;
        struct qualifiedCode {
            CodingSchemeId coding_scheme_id;
            ConceptCode a_code;
        };
        struct QualifiedPersonIdInfo {
            QualifiedCode a_qualified_code;
            QualifiedPersonId id;
    }


#endif // _PERSON_ID_TRAITS_IDL_
```

The **PersonIdTraits** module contains definitions for traits that are needed but are not defined by the HL7 Version 2.3 or vCard Version 2.1 standards. Most of these are for defining other IDs a person may have where these IDs can be characterized in the format of this specification.

NULL_TRAIT, NullTraitType

This trait is used in places where a trait or trait name must be passed but no valid value exists.

INTERNAL_ID, InternalIdType

The **InternalId**, if used, is always set to the **PersonID** itself. It is specified in the **supported_traits** as read-only and searchable but not mandatory. The reason for including it as a trait is to allow matching on it. This is the only way the ID can be matched on when only part of the ID is known. For example, this could occur if a person’s ID card has a tear or only part of the digits are remembered by the person.

MERGED_IDS, MergedIdsType

This trait indicates the set of other IDs a person may have. The IDs listed are from the same ID Domain as the ID this trait is bound to. It has the special semantics that the IDs listed are in the Deprecated **IdState** and the **preferred_id** on their **IdInfo** references the ID this trait is bound to. The setting of this trait is controlled by the PIDS implementation

in order to maintain consistency with the back references from the IDs listed; therefore, this trait is always read-only when supported by a PIDS.

DUPLICATE_IDS, DuplicateIdsType

This trait indicates the set of other IDs a person may have. The IDs listed are from the same ID Domain as the ID this trait is bound to. This is more general purpose than **MergedIds**. The IDs listed in this trait do not have to be deprecated or merged.

CORRELATED_IDS, CorrelatedIdsType

This trait indicates the set of other IDs a person may have. The IDs listed are from a different ID Domain than the ID Domain of the ID this trait is bound to. This trait has special semantics in that it refers to IDs that have been correlated by the **CorrelationMgr**; therefore, this trait is always read-only when supported by a PIDS.

EXTERNAL_IDS, ExternalIdsType

This trait indicates the set of other IDs a person may have. The IDs listed may be from a different ID Domain than the ID Domain of the ID this trait is bound to. This trait is more general purpose than **CorrelatedIds**. The IDs listed in this trait may be set by mechanisms other than automatic correlation.

NATIONAL_HEALTH_IDS, NationalHealthIdsType

The **NationalHeathId** trait is added in this module because the HL7 2.3 and vCard 2.1 standards do not have a corresponding trait. This trait is important in countries that have a national healthcare ID. The “country_code” field indicates the name of the country issuing the IDs. The ISO 3166 two-letter country codes or the telephone numeric country codes are to be used. The value for this trait consists of a sequence since it is possible for a person to have IDs in multiple countries or accidently get multiple IDs in the same country.

4.2 HL7Version2_3 Module

```
//File: HL7Version2_3.idl

#ifndef _HL7_VERSION_2_3_IDL_
#define _HL7_VERSION _HL7_VERSION_2_3_IDL_

#include <orb.idl>
#include <PersonIdService.idl>

#pragma prefix "omg.org"

module HL7Version2_3
{
    typedef PersonIdService::TraitName PIDFieldName;
    typedef PersonIdService::TraitName RepPIDFieldName
    typedef string PIDFieldValue;

    const RepPIDFieldName PATIENT_NAME                  = "HL7/PatientName";
```

```
    const PIDFieldName MOTHERS_MAIDEN_NAME          = "HL7/MothersMaidenName";
    const PIDFieldName DATE_TIME_OF_BIRTH           = "HL7/DateTimeofBirth";
    const PIDFieldName SEX                          = "HL7/Sex";
    const RepPIDFieldName PATIENT_ALIAS             = "HL7/PatientAlias";
    const PIDFieldName RACE                         = "HL7/Race";
    const RepPIDFieldName PATIENT_ADDRESS           = "HL7/PatientAddress";
    const PIDFieldName COUNTY_CODE                  = "HL7/CountyCode";
    const RepPIDFieldName PHONE_NUMBER_HOME         = "HL7/PhoneNumber_Home";
    const RepPIDFieldName PHONE_NUMBER_BUSINESS     = "HL7/PhoneNumber_Business";
    const PIDFieldName PRIMARY_LANGUAGE             = "HL7/PrimaryLanguage";
    const PIDFieldName MARITAL_STATUS               = "HL7/MaritalStatus";
    const PIDFieldName RELIGION                     = "HL7/Religion";
    const PIDFieldName PATIENT_ACCOUNT_NUMBER       = "HL7/PatientAccountNumber";
    const PIDFieldName SSN_NUMBER                   = "HL7/SSNNumber";
    const PIDFieldName DRIVERS_LICENSE_NUMBER       = "HL7/DriversLicenseNumber";
    const RepPIDFieldName MOTHERS_IDENTIFIER        = "HL7/MothersIdentifier";
    const PIDFieldName ETHNIC_GROUP                 = "HL7/EthnicGroup";
    const PIDFieldName BIRTH_PLACE                  = "HL7/BirthPlace";
    const PIDFieldName MULTIPLE_BIRTH_INDICATOR     = "HL7/MultipleBirthIndicator";
    const PIDFieldName BIRTH_ORDER                  = "HL7/BirthOrder";
    const RepPIDFieldName CITIZENSHIP               = "HL7/Citizenship";
    const PIDFieldName VETERANS_MILITARY_STATUS     = "HL7/VeteransMilitaryStatus";
    const PIDFieldName NATIONALITY                  = "HL7/Nationality";
    const PIDFieldName PATIENT_DEATH_DATE_AND_TIME  = "HL7/PatientDeathDateandTime";
    const PIDFieldName PATIENT_DEATH_INDICATOR      = "HL7/PatientDeathIndicator";
};

#endif //_HL7_VERSION_2_3_IDL_
```

The **HL7Version2_3** module defines the standard trait names for using trait information from the Health Level Seven (HL7) standard. The traits correspond to fields 5-30 in the PID segment as defined in Version 2.3 of the HL7 standard. The values for the traits are strings (**HL7Version2_3::PIDFieldValue**) formatted according to the rules specified by the HL7 standard.

The values for these HL7 PID traits contain the exact semantics as the corresponding fields in the HL7 Version 2.3 PID segment. For example, where the “CE” data type is used any HL7 recommended value sets are also recommended when using them with PIDS.

The trait names (**HL7Version2_3::PIDFieldName**) include the complete **QualifiedNameStr** which has a registration authority of OTHER and the authority ID of “HL7.”

Since the PIDS specification recommends to use the data types and the formats for the traits in the PID segment to conform to the HL7 formats and data types, we do not have to add a new data type of sequence of string.

Many of these traits from the HL7 PID segment are demographic traits instead of identifying traits. The difference between demographic traits and identifying traits is often dependent on the environment in which they are being used. It was decided to include all of these PID segment traits and let the PIDS implementors choose the ones that are considered identifying traits for their service.

Note – HL7 is working on version 3.0 of its standard. Once it is finished the list of traits may be updated or a new list created via the standard OMG processes.

4.2.1 HL7 Link and Unlink Events

The HL7 standard defines distinct “trigger events” for merge and unmerge, and for link and unlink. The distinction between the two pairs is as follows: While “merge” deactivates one identifier in favor of another, link simply asserts that the identifiers involved represent the same real-world person - commonly referred to as “duplicates.”

In this specification, the **merge_ids** and **unmerge_ids** operations are semantically equivalent to their HL7 counterparts, but there are no explicit PIDS operations corresponding to the HL7 link and unlink events. However, the PIDS fully supports the link and unlink semantics by means of the **ProfileAccess: update_and_clear_traits** operation. The client simply asserts or clears values for the **DuplicateIds** trait. Note that an **IdMgr** implementation should clear **DuplicateIDs** not only when a client clears them explicitly using **update_and_clear_traits**, but also when a merge is performed on a pair of identifiers that is on record as being duplicates.

4.3 vCardTraits Module

```
//File: vCardVersion2_1.idl

#ifndef _V_CARD_VERSION_2_1_IDL_
#define _V_CARD_VERSION_2_1_IDL_

#include <orb.idl>
#include <PersonIdService.idl>

#pragma prefix "omg.org"

module vCardVersion2_1
{
    typedef PersonIdService::TraitName PropertyName;
    typedef string PropertyValue;

    const PropertyName FORMATTED_NAME           = "vCard/FN";
    const PropertyName NAME                     = "vCard/N";
    const PropertyName PHOTOGRAPH               = "vCard/PHOTO";
    const PropertyName BIRTHDAY                 = "vCard/BDAY";
    const PropertyName ADDRESS                  = "vCard/ADR";
    const PropertyName HOME_ADDRESS             = "vCard/ADR;HOME";
    const PropertyName WORK_ADDRESS             = "vCard/ADR;WORK";
    const PropertyName TELEPHONE                = "vCard/TEL";
    const PropertyName PREFERRED_TELEPHONE      = "vCard/TEL;PREF";
    const PropertyName HOME_TELEPHONE           = "vCard/TEL;HOME";
    const PropertyName WORK_TELEPHONE           = "vCard/TEL;WORK";
    const PropertyName VOICE_TELEPHONE          = "vCard/TEL;VOICE";
    const PropertyName FAX_TELEPHONE            = "vCard/TEL;FAX";
    const PropertyName MESSAGE_TELEPHONE        = "vCard/TEL;MSG";
```

```
    const PropertyName CELLULAR_TELEPHONE              = "vCard/TEL;CELL";
    const PropertyName BULLETIN_BOARD_TELEPHONE        =" vCard/TEL;BBS";
    const PropertyName MODEM_TELEPHONE                 ="vCard/TEL;MODEM";
    const PropertyName CAR_TELEPHONE                   = "vCard/TEL;CAR";
    const PropertyName ELECTRONIC_MAIL                 = "vCard/EMAIL";
    const PropertyName GEOGRAPHIC_POSITION             = "vCard/GEO";
    const PropertyName TITLE                           = "vCard/TITLE";
    const PropertyName ORGANIZATION                    = "vCard/ORG";
    const PropertyName SOUND_ANNOTATION                = "vCard/SOUND";
    const PropertyName UNIFORM_RESOURCE_LOCATOR        = "vCard/URL";
};

#endif //_V_CARD_VERSION_2_1_IDL_
```

The **vCardVersion2_1** module defines the standard trait names for using trait information from the vCard standard. The traits correspond to the properties as defined in Version 2.1 of the vCard standard. The values for the traits are strings (**vCardVersion2_1::PropertyValue**) formatted according to the rules specified by the vCard standard.

Only the properties in vCard that relate to this specification are included. The appropriate properties with parameters are also included.

The trait names (**vCardVersion2_1::PropertyName**) include the complete **QualifiedNameStr** which has a registration authority of OTHER and the authority ID of “vCard.”

Naming/Trader Interoperation 5

Contents

This chapter contains the following topics.

The Naming and Trader Services are used in multiple ways by the PIDS specification. One way is to define Trader Service Types explicitly for PIDS and some naming conventions for PIDS-related objects. The other is that every PIDS component has the ability to expose Naming and Trader services that are local to the component to provide explicate federation of PIDS implementations.

5.1 Naming Service

The following names are to be used for the “kind” field in **CosNaming::NameComponent**.

- **“IdentificationComponent”** - Generic type for a PIDS component.
- **“Simple PIDS”** - A PIDS component that meets the conformance class of the same name.
- **“Sequential Access PIDS”** - A PIDS component that meets the conformance class of the same name.
- **“Identity Access PIDS”** - A PIDS component that meets the conformance class of the same name.

- **“ID Domain Manager PIDS”** - A PIDS component that meets the conformance class of the same name.
- **“Correlation PIDS”** - A PIDS component that meets the conformance class of the same name.

Naming contexts are freestanding nodes in the directory hierarchy unless they are associated with a particular system or component in some way. The use of the naming context, as referenced from an **IdentificationComponent**, has specific semantics. The naming context referenced from a PIDS component is the root of a naming tree relative to that component. It is a way for the component to publish references to other objects it is associated with or knows about. The naming directory for which it is the root can be used as a general naming service as well.

The following directory names have special meaning for PIDS components:

- **Source ID Domains** - Components that do correlation over other ID Domains will have object references to PIDS components that manage those ID Domains (if they exist) in this directory. It is suggested that the sub ID Domains be referenced by their stringified ID Domain name as it provides a simple way to search for ID Domains when a trader component is not available. They may also be referenced by a logical name for easier identification by users and by their component name.
- **Correlating ID Domains** - Components that are being correlated by other components can publish references to these correlating components in this directory.
- **ID Domain Components** - PIDS components that manage an ID Domain will have references to other PIDS components within the ID Domain in this directory. It is suggested that the ID Domain residents be referenced by their stringified component name as it provides a simple way to search for them when a trader component is not available. They may also be referenced by a logical name for easier identification by users.
- **ID Using Services** - This directory is a place for a component to have references to services it provides that use IDs from its ID Domain.

The following names for objects located in the root of the tree referenced by a PIDS component are also reserved.

- Trait Information - If this object exists, it is an implementation of the Lexicon Query Service (LQS) that contains the traits supported by this PIDS component as concepts.
- Trader components are typically stand-alone services installed in an enterprise. The reference to the trader component from a PIDS component has special semantics. The trader component is used for searching for other PIDS components as well as other object services. The Trader referenced by a PIDS component knows of the PIDS service types defined in this specification.

5.2 Trader Service

The following definitions are Service Types defined for PIDS components for use by the Trader Service.

```
service IdentificationComponent {
        interface IdentificationComponent;
        mandatory readonly property string domain_name;
        mandatory readonly property StringSeq interfaces_implemented;
        mandatory readonly property StringSeq conformance_classes;
        mandatory readonly property string component_name;
        mandatory readonly property string component_version;
        mandatory readonly property StringSeq supported_traits;
        mandatory readonly property StringSeq read_only_traits;
        mandatory readonly property StringSeq mandatory_traits;
        mandatory readonly property StringSeq searchable_traits;
        readonly property StringSeq source_domains;
};
```

Since all PIDS implement the **IdentificationComponent** only one Trader Service type is needed, which is also called "**IdentificationComponent**." The **IdentificationComponent** interface has attributes for the common characteristics for all PIDS. These are used as properties for the **IdentificationComponent** service type. One additional property is specified as well which comes from an attribute from one of the derived interfaces. The stringified versions of the attributes are used for properties since the standard Trader constraint language does not provide a way to filter on user-defined types.

5.2.1 IdentificationComponent Service

The interface type returned from the Trader Service for this service type is an **IdentificationComponent**. All except one of the properties are mandatory. These are found on all **IdentificationComponent** interface implementations.

domain_name

The **domain_name** property contains the information from the **domain_name** attribute of the **IdentificationComponent** interface. It is formatted as specified for **NamingAuthority::AuthorityIdStr**.

interfaces_implemented

This sequence contains the names of the interfaces the component has references to. This includes PIDS-specific interfaces such as "**PersonIdService::ProfileAccess**" and other interfaces such as "**CosNaming::NamingContext**." The names are fully qualified names which include the module name.

conformance_classes

This sequence contains the conformance classes the implementation supports. The strings are identical to the way they are spelled and capitalized in the definition of the conformance classes for PIDS.

component_name

This property contains the "**the_name**" field of the **component_name** attribute. It is a

stringified version of the name in the format as a **NamingAuthority::QualifiedNameStr**.

component_version

This property contains the "**the_version**" field from the **component_name** attribute.

supported_traits

This property contains a sequence with all of the **TraitNames** that are in the **supported_traits** attribute.

read_only_traits

This property contains a sequence with all of the **TraitNames** that are in the **supported_traits** attribute that have the **read_only** field set to true.

mandatory_traits

This property contains a sequence with all of the **TraitNames** that are in the **supported_traits** attribute that have the mandatory field set to true.

searchable_traits

This property contains a sequence with all of the **TraitNames** that are in the **supported_traits** attribute that have the searchable field set to true.

source_domains

This is the only optional property. It only applies to PIDS that implement the **CorrelationMgr** interface. The property contains the stringified **DomainName** for the source ID Domains being correlated.

Conformance Classes 6

The following interfaces are programmatic reference points for testing conformance. Conformance indicates implementing all of the attributes and operations for those interfaces and the specified behavior, and raising exceptions as specified. For the operations that may raise the PersonIdService::NotImplemented exception, a valid implementation of the operation may raise this exception. All other operations must perform as specified within this specification.

- **IdentificationComponent**
- **ProfileAccess**
- **IdentityAccess**
- **SequentialAccess**
- **IdentifyPerson**
- **IdMgr**
- **CorrelationMgr**
- **EventComponent**

The following taxonomy is defined for specific conformance classes of PIDS service implementations. An implementation claiming conformance to any of these classes must conform to all of the interfaces specified for that class. An implementation may claim conformance to multiple conformance classes as long as it conforms to each one it claims.

An implementation claiming conformance must also follow the rules in the **NamingAuthority** module for creating names that derive from those types.

All conforming implementations must support at least one trait as specified on their **supported_traits** attribute.

Implementations that use the traits defined in the **PersonIdTraits**, HL7Version2_3 and **vCardVersion2_1** modules must maintain the semantics defined in those modules to be considered conforming.

An implementation claiming conformance must implement the semantics defined for **IdentificationComponent** such as consistency between all the interfaces implemented. They must also maintain the semantic mapping between attributes on the **IdentificationComponent** and the other interfaces implemented.

Each row in the table below includes the specification for a different conformance class. The columns represent the interfaces on the **IdentificationComponent**. A "*" symbol in a column indicates the conformance class in that row includes the interface of that column.

Conformance Class	Identify Person	Profile Access	Sequential Access	IDMgr.	Identity Access	CorrelationMgr
Simple PIDS	*	*				
Sequential Access PIDS	*	*	*			
ID Domain Mgr PIDS	*	*		*		
Identity Access PIDS	*				*	
Correlation PIDS						*

- **"Simple PIDS"** - Provides the basic operations to access profiles from an ID and to match potential IDs given some traits.
- **"Sequential Access PIDS"** - Adds the ability to scroll through the set of IDs sequentially.
- **"ID Domain Mgr PIDS"** - Adds the ability to create new IDs and modify the states of IDs that were already created.
- **"Identity Access PIDS"** - An alternative to the Simple PIDS in that it provides the same functionality but uses the **IdentityAccess** interface instead of the **ProfileAccess**. It is intended for implementations that have different access policies for each ID, for example allowing people to control access to their own profile.
- **"Correlation PIDS"** - The **supported_traits** attribute contains at least the following three traits: **HL7:PatientName**; **HL7:DateTimeOfBirth**; **HL7:Sex**. This conformance class contains a single interface as it provides functionality by itself; however, it can be mixed in with the other conformance classes for additional functionality.

References *A*

A.1 List of References

Health Level Seven (HL7) Standard, version 2.3, 1997.

Versit Consortium. VCard - The Electronic Business Card, version 2.1, September 18, 1996.

American Dental Association, "Proposed ANSI/ADA Specification No 1000 Standard Clinical Data Architecture for the Structure and Content of a Computer-based Patient Record. Part 1000.1 Individual Identification", ASC MD156, August 1997.

ASTM E1239, Standard Guide for Description of Reservation/Registration-Admission, Discharge, Transfer (R-ADT) Systems for Automated Patient Care Information Systems, Committee E-31 on Computerized Systems, Subcommittee E31.19 on Vocabulary for Computer-Based Patient Records-Content and Structure, West Conshohocken, PA: ASTM, January 15, 1994.

ASTM E1714. Guide for the Properties of a Universal Healthcare Identifier. Committee E-31 on Computerized Systems, Subcommittee E31.12 on Medical Records. West Conshohocken, PA: ASTM, Aug. 15, 1995.

ASTM E1385. Guide for global environments for RADT.

ASTM E1715. RADT data model.

CPRI, 1996a. Action Plan for Implementing a Universal Patient Identifier, Draft Version 1.0. Schaumburg, IL: Computer-based Patient Record Institute, May.

ICSI, 1995. Data Communication Standard: Patient Identifier, September 1995.

HIN, 1997. The Essential Medical Data Set (EMDS), January 8, 1997.

UK National Health System Common Data Model.

A

Complete OMG IDL B

B.1 Complete IDL Listing

```
//File: PersonIdService.idl

#ifndef _PERSON_ID_SERVICE_IDL_
#define _PERSON_ID_SERVICE_IDL_

#include <orb.idl>
#include <NamingAuthority.idl>
#include <Naming.idl>
#include <Trading.idl>

#pragma prefix "omg.org"

module PersonIdService
{
#    pragma version Person Identification Service 1.1

        // ----------------------------------------------------------------
        // Common Data Types
        //
        typedef NamingAuthority::AuthorityId DomainName;
        typedef sequence< DomainName > DomainNameSeq;

        typedef NamingAuthority::LocalName PersonId;
        typedef sequence< PersonId > PersonIdSeq;

        struct QualifiedPersonId {
            DomainName domain;
            PersonId id;
        };
        typedef sequence< QualifiedPersonId > QualifiedPersonIdSeq;

        typedef NamingAuthority::QualifiedNameStr TraitName;
        typedef sequence< TraitName > TraitNameSeq;
```

```
typedef any TraitValue;
struct Trait {
      TraitName name;
      TraitValue value;
};
typedef sequence< Trait > TraitSeq;
typedef TraitSeq Profile;
typedef sequence< Profile > ProfileSeq;

enum IdState { UNKNOWN, INVALID, TEMPORARY, PERMANENT, DEACTIVATED };
typedef sequence<IdState> IdStateSeq;
struct IdInfo {
      PersonId id;
      IdState state;
      PersonId preferred_id;
};
typedef sequence<IdInfo> IdInfoSeq;


// ------------------------------------------------------------
// Miscellaneous Data Types
//

typedef string ComponentVersion;
struct ComponentName {
      NamingAuthority::QualifiedName name;
      ComponentVersion version;
};

struct TraitSpec {
      TraitName trait;
      boolean mandatory;
      boolean read_only;
      boolean searchable;
};
typedef sequence< TraitSpec > TraitSpecSeq;

enum HowManyTraits { NO_TRAITS, SOME_TRAITS, ALL_TRAITS };
union SpecifiedTraits switch ( HowManyTraits )
{
      case SOME_TRAITS: TraitNameSeq traits;
};

struct TaggedProfile {
      PersonId id;
      PersonIdService::Profile profile;
};
typedef sequence<TaggedProfile> TaggedProfileSeq;

struct QualifiedTaggedProfile {
      QualifiedPersonId id;
      PersonIdService::Profile profile;
};
typedef sequence<QualifiedTaggedProfile> QualifiedTaggedProfileSeq;
```

```
struct ProfileUpdate {
      PersonId id;
      TraitNameSeq del_list;
      TraitSeq modify_list;
};
typedef sequence< ProfileUpdate > ProfileUpdateSeq;

struct MergeStruct {
      PersonId id;
      PersonId preferred_id;
};
typedef sequence< MergeStruct > MergeStructSeq;

struct TraitSelector {
      PersonIdService::Trait trait;
      float weight;
};
typedef sequence<TraitSelector> TraitSelectorSeq;

struct Candidate {
      PersonId id;
      float confidence;
      PersonIdService::Profile profile;
};
typedef sequence<Candidate> CandidateSeq;

interface CandidateIterator {
      unsigned long max_left();

      boolean next_n(
            in unsigned long n,
            out CandidateSeq ids );

      void destroy();
};

typedef unsigned long Index;
typedef sequence< Index > IndexSeq;

enum ExceptionReason {
      UNKNOWN_TRAITS,
      DUPLICATE_TRAITS,
      WRONG_TRAIT_FORMAT,
      REQUIRED_TRAITS,
      READONLY_TRAITS,
      CANNOT_REMOVE,
      MODIFY_OR_DELETE
};

struct MultipleFailure {
      Index the_index;
      ExceptionReason reason;
      TraitNameSeq traits;
};
typedef sequence< MultipleFailure > MultipleFailureSeq;
```

```
interface Identity;
typedef sequence< Identity > IdentitySeq;


// ------------------------------------------------------------------
// Exceptions
//

exception InvalidId { IdInfo id_info; };
exception InvalidIds { IdInfoSeq id_info; };
exception DuplicateIds { PersonIdSeq ids; };
exception UnknownTraits { TraitNameSeq traits; };
exception DuplicateTraits { TraitNameSeq traits; };
exception WrongTraitFormat { TraitNameSeq traits; };
exception InvalidStates {};
exception TooMany { unsigned long estimated_max; };
exception MultipleTraits { MultipleFailureSeq failures; };

exception ReadOnlyTraits { TraitNameSeq traits; };
exception CannotRemove { TraitNameSeq traits; };
exception ModifyOrDelete { MultipleFailureSeq failures; };
exception NotImplemented {};

exception InvalidWeight {};
exception CannotSearchOn { TraitNameSeq traits; };

exception IdsExist { IndexSeq indices; };
exception RequiredTraits { TraitNameSeq traits; };
exception ProfilesExist { IndexSeq indices; };
exception DuplicateProfiles { IndexSeq indices; };

exception DomainsNotKnown { DomainNameSeq domain_names; };
exception IdsNotKnown { QualifiedPersonIdSeq ids; };


// ------------------------------------------------------------
// IdentificationComponent
//

interface ProfileAccess;
interface SequentialAccess;
interface IdentityAccess;
interface IdentifyPerson;
interface IdMgr;
interface CorrelationMgr;

interface IdentificationComponent
{
    readonly attribute DomainName                   domain_name;
    readonly attribute ComponentName                component_name;
    readonly attribute TraitSpecSeq                 supported_traits;

    readonly attribute IdentifyPerson               identify_person;
    readonly attribute ProfileAccess                profile_access;
```

```
        readonly attribute SequentialAccess                 sequential_access;
        readonly attribute IdentityAccess                   identity_access;

        readonly attribute IdMgr                            id_mgr;

        readonly attribute CorrelationMgr                   correlation_mgr;
        readonly attribute CosNaming::NamingContext         naming_context;
        readonly attribute CosTrading::TraderComponents     trader_components;
            void get_supported properties(
                in TraitName name,
                out CosPropertyService::Properties trait_defs)
                raises(
                UnknownTraits
};
};


// ---------------------------------------------------------------
// IdentifyPerson
//

interface IdentifyPerson :
IdentificationComponent
{
        void find_candidates(
                in TraitSelectorSeq profile_selector,
                in IdStateSeq states_of_interest,
                in float confidence_threshold,
                in unsigned long sequence_max,
                in unsigned long iterator_max,
                in SpecifiedTraits traits_requested,
                out CandidateSeq returned_sequence,
                out CandidateIterator returned_iterator )
            raises (
                TooMany,
                UnknownTraits,
                WrongTraitFormat,
                CannotSearchOn,
                DuplicateTraits,
                InvalidStates,
                InvalidWeight );
};

// ---------------------------------------------------------------
// ProfileAccess
//

interface ProfileAccess :
        IdentificationComponent
{
        TraitNameSeq get_traits_known(
                in PersonId id  )
            raises (
                InvalidId );
```

```
        Profile get_profile(
                in PersonId id,
                in SpecifiedTraits traits_requested )
            raises (
                InvalidId,
                UnknownTraits,
                DuplicateTraits );

        TaggedProfileSeq get_profile_list(
                in PersonIdSeq ids,
                in SpecifiedTraits traits_requested )
            raises (
                TooMany,
                InvalidIds,
                DuplicateIds,
                UnknownTraits,
                DuplicateTraits );

        TaggedProfileSeq get_deactivated_profile_list(
                in PersonIdSeq ids,
                in SpecifiedTraits traits_requested )
            raises (
                NotImplemented,
                InvalidIds,
                DuplicateIds,
                UnknownTraits,
                DuplicateTraits );

        void update_and_clear_traits(
                in ProfileUpdateSeq profile_update_spec )
            raises (
                InvalidIds,
                DuplicateIds,
                NotImplemented,
                MultipleTraits );

        IdInfoSeq get_id_info(
                in PersonIdSeq ids )
            raises (
                TooMany,
                DuplicateIds );
};

// ------------------------------------------------------------------
// SequentialAccess
//

interface SequentialAccess :
        IdentificationComponent
{
        unsigned long id_count_per_state(
                in IdStateSeq states_of_interest )
            raises (
                InvalidStates );
```

```
        TaggedProfileSeq get_all_ids_by_state(
                in SpecifiedTraits traits_requested,
                in IdStateSeq states_of_interest )
            raises (
                TooMany,
                UnknownTraits,
                DuplicateTraits,
                InvalidStates );

        TaggedProfileSeq get_first_ids(
                in unsigned long how_many,
                in IdStateSeq states_of_interest,
                in SpecifiedTraits traits_requested )
            raises (
                TooMany,
                UnknownTraits,
                DuplicateTraits,
                InvalidStates );

        TaggedProfileSeq get_last_ids(
                in unsigned long how_many,
                in IdStateSeq states_of_interest,
                in SpecifiedTraits traits_requested )
            raises (
                TooMany,
                UnknownTraits,
                DuplicateTraits,
                InvalidStates );

        TaggedProfileSeq get_next_ids(
                in PersonId reference_id,
                in unsigned long how_many,
                in IdStateSeq states_of_interest,
                in SpecifiedTraits traits_requested )
            raises (
                TooMany,
                InvalidId,
                UnknownTraits,
                DuplicateTraits,
                InvalidStates );

        TaggedProfileSeq get_previous_ids(
                in PersonId reference_id,
                in unsigned long how_many,
                in IdStateSeq states_of_interest,
                in SpecifiedTraits traits_requested )
            raises (
                TooMany,
                InvalidId,
                UnknownTraits,
                DuplicateTraits,
                InvalidStates );
    };
```

```
// ------------------------------------------------------------------
// IdentityAccess
//

interface IdentityAccess :
      IdentificationComponent
{
      Identity get_identity_object(
            in PersonId id )
         raises (
            InvalidId );

      IdentitySeq get_identity_objects(
            in PersonIdSeq ids )
         raises (
            InvalidIds );
};

interface Identity
{
      readonly attribute IdentificationComponent source_component;
      readonly attribute IdInfo id_info;
      readonly attribute TraitNameSeq traits_with_values;
      readonly attribute long trait_value_count;

      Trait get_trait(
            in TraitName trait_requested )
         raises (
            UnknownTraits );

      Profile get_profile(
            in SpecifiedTraits traits_requested )
         raises (
            UnknownTraits,
            DuplicateTraits );

      Profile get_deactivated_profile(
            in SpecifiedTraits traits_requested )
         raises (
            NotImplemented,
            UnknownTraits,
            DuplicateTraits );

      void update_and_clear_traits(
            in ProfileUpdate profile_update_spec )
         raises (
            NotImplemented,
            UnknownTraits,
            WrongTraitFormat,
            ModifyOrDelete,
            ReadOnlyTraits,
            CannotRemove,
            DuplicateTraits );
```

```
        void done();
};

// ------------------------------------------------------------
// IdMgr
//

interface IdMgr :
        IdentificationComponent
{
        PersonIdSeq register_new_ids(
                in ProfileSeq profiles_to_register )
            raises (
                ProfilesExist,
                DuplicateProfiles,
                MultipleTraits );

        PersonIdSeq find_or_register_ids(
                in ProfileSeq profiles_to_register )
            raises (
                DuplicateProfiles,
                MultipleTraits );

        void register_these_ids(
                in TaggedProfileSeq profiles_to_register )
            raises (
                NotImplemented,
                IdsExist,
                DuplicateIds,
                ProfilesExist,
                DuplicateProfiles,
                MultipleTraits );

        PersonIdSeq create_temporary_ids(
                in ProfileSeq profiles_to_register )
            raises (
                MultipleTraits );

        PersonIdSeq make_ids_permanent(
                in PersonIdSeq ids_to_modify )
            raises (
                InvalidIds,
                DuplicateIds,
                RequiredTraits );

        IdInfoSeq merge_ids(
                in MergeStructSeq ids_to_merge )
            raises (
                InvalidIds,
                DuplicateIds );

        IdInfoSeq unmerge_ids(
                in PersonIdSeq ids_to_unmerge )
            raises (
                InvalidIds,
```

```
                    DuplicateIds );

            IdInfoSeq deprecate_ids(
                    in PersonIdSeq ids_to_deprecate )
                raises (
                    InvalidIds,
                    DuplicateIds );
        };


        // ------------------------------------------------------------
        // CorrelationMgr
        //

        interface CorrelationMgr :
            IdentificationComponent
{
            readonly attribute DomainNameSeq source_domains;

            void load_profiles(
            in QualifiedTaggedProfileSeq tagged_profiles )
                raises (
                    UnknownTraits,
                    WrongTraitFormat,
                    DomainsNotKnown );

            QualifiedPersonIdSeq get_corresponding_ids(
                    in QualifiedPersonId from_id,
                    in DomainNameSeq to_domains )
                raises (
                    DomainsNotKnown,
                    IdsNotKnown );
        };

            PersonIdSeq find_or_register_ids(
                    in QualifiedTaggedProfileSeq tagged_profiles )
                raises (
                    MultipleTraits,
                    DomainsNotKnown,
                    NotImplemented );
        };
    };


#endif // _PERSON_ID_SERVICE_IDL_



//File: NamingAuthority.idl

#ifndef _NAMING_AUTHORITY_IDL_
#define _NAMING_AUTHORITY_IDL_

#include <orb.idl>
```

```
#pragma prefix "omg.org "

module NamingAuthority
{
        enum RegistrationAuthority {
            OTHER,
            ISO,
            DNS,
            IDL,
            DCE };

        typedef string NamingEntity;

        struct AuthorityId {
            RegistrationAuthority   authority;
            NamingEntity            naming_entity;
        };
        typedef string AuthorityIdStr;

        typedef string LocalName;
        struct QualifiedName {
            AuthorityId authority_id;
            LocalName local_name;
        };
        typedef string QualifiedNameStr;

        exception InvalidInput {};

        interface translation_library
        {
            AuthorityIdStr authority_to_str(
                    in AuthorityId authority )
                raises(
                    InvalidInput );

            AuthorityId str_to_authority(
                    in AuthorityIdStr authority_str )
                raises(
                    InvalidInput );

            QualifiedNameStr qualified_name_to_str(
                    in QualifiedName qualified_name )
                raises(
                    InvalidInput );

            QualifiedName str_to_qualified_name(
                    in QualifiedNameStr qualified_name_str )
                raises(
                    InvalidInput );
        };
};

#endif // _NAMING_AUTHORITY_IDL_
```

```
//File: PersonIdTraits.idl

#ifndef _PERSON_ID_TRAITS_IDL_
#define _PERSON_ID_TRAITS_IDL_

#include <orb.idl>
#include <PersonIdService.idl>

#pragma prefix "omg.org"

module PersonIdTraits
{
        const PersonIdService::TraitName NULL_TRAIT = "";
        typedef any NullTraitType;  // set to tk_null

        const PersonIdService::TraitName INTERNAL_ID = "PIDS/InternalId";
        typedef PersonIdService::PersonId InternalIdType;

        const PersonIdService::TraitName MERGED_IDS = "PIDS/MergedIds";
        typedef PersonIdService::PersonIdSeq MergedIdsType;

        const PersonIdService::TraitName DUPLICATE_IDS = "PIDS/DuplicateIds";
        typedef PersonIdService::PersonIdSeq DuplicateIdsType;

        const PersonIdService::TraitName CORRELATED_IDS = "PIDS/CorrelatedIds";
        typedef PersonIdService::QualifiedPersonIdSeq CorrelatedIdsType;

        const PersonIdService::TraitName EXTERNAL_IDS = "PIDS/ExternalIds";
        typedef PersonIdService::QualifiedPersonIdSeq ExternalIdsType;

        const PersonIdService::TraitName NATIONAL_HEALTH_IDS = "PIDS/NationalHealthId";
        struct NationalHealthIdStruct {
            string country_code;
            PersonIdService::PersonId id;
        };
        typedef sequence< NationalHealthIdStruct > NationalHealthIdStructSeq;
        typedef NationalHealthIdStructSeq NationalHealthIdsType;
};


#endif // _PERSON_ID_TRAITS_IDL_


//File: HL7Version2_3.idl

#ifndef _HL7_VERSION_2_3_IDL_
#define _HL7_VERSION_2_3_IDL_

#include <orb.idl>
#include <PersonIdService.idl>

#pragma prefix "omg.org"

module HL7Version2_3
{
```

```
        typedef PersonIdService::TraitName PIDFieldName;
        typedef string PIDFieldValue;

        const PIDFieldName PATIENT_NAME                 = "HL7/PatientName";
        const PIDFieldName MOTHERS_MAIDEN_NAME          = "HL7/MothersMaidenName";
        const PIDFieldName DATE_TIME_OF_BIRTH           = "HL7/DateTimeofBirth";
        const PIDFieldName SEX                          = "HL7/Sex";
        const PIDFieldName PATIENT_ALIAS                = "HL7/PatientAlias";
        const PIDFieldName RACE                         = "HL7/Race";
        const PIDFieldName PATIENT_ADDRESS              = "HL7/PatientAddress";
        const PIDFieldName COUNTY_CODE                  = "HL7/CountyCode";
        const PIDFieldName PHONE_NUMBER_HOME.           = "HL7/PhoneNumber_Home";
        const PIDFieldName PHONE_NUMBER_BUSINESS        = "HL7/PhoneNumber_Business";
        const PIDFieldName PRIMARY_LANGUAGE             = "HL7/PrimaryLanguage";
        const PIDFieldName MARITAL_STATUS               = "HL7/MaritalStatus";
        const PIDFieldName RELIGION                     = "HL7/Religion";
        const PIDFieldName PATIENT_ACCOUNT_NUMBER       = "HL7/PatientAcountnumber";
        const PIDFieldName SSN_NUMBER                   = "HL7/SSNNumber";
        const PIDFieldName DRIVERS_LICENSE_NUMBER       = "HL7/DriversLicenseNumber";
        const PIDFieldName MOTHERS_IDENTIFIER           = "HL7/MothersIdentifier";
        const PIDFieldName ETHNIC_GROUP                 = "HL7/EthnicGroup";
        const PIDFieldName BIRTH_PLACE                  = "HL7/BirthPlace";
        const PIDFieldName MULTIPLE_BIRTH_INDICATOR     = "HL7/MultipleBirthIndicator";
        const PIDFieldName BIRTH_ORDER                  = "HL7/BirthOrder";
        const PIDFieldName CITIZENSHIP                  = "HL7/Citizenship";
        const PIDFieldName VETERANS_MILITARY_STATUS     = "HL7/VeteransMilitaryStatus";
        const PIDFieldName NATIONALITY                  = "HL7/Nationality";
        const PIDFieldName PATIENT_DEATH_DATE_AND_TIME= "HL7/PatientDeathDateandTime";
        const PIDFieldName PATIENT_DEATH_INDICATOR      = "HL7/PatientDeathIndicator";
};

#endif //_HL7_VERSION_2_3_IDL_


//File: vCardVersion2_1.idl

#ifndef _V_CARD_VERSION_2_1_IDL_
#define _V_CARD_VERSION_2_1_IDL_

#include <orb.idl>
#include <PersonIdService.idl>

#pragma prefix "omg.org"

module vCardVersion2_1
{
        typedef PersonIdService::TraitName PropertyName;
        typedef string PropertyValue;

        const PropertyName FORMATTED_NAME               = "vCard/FN";
        const PropertyName NAME                         = "vCard/N";
        const PropertyName PHOTOGRAPH                   = "vCard/PHOTO";
        const PropertyName BIRTHDAY                     = "vCard/BDAY";
        const PropertyName ADDRESS                      = "vCard/ADR";
        const PropertyName HOME_ADDRESS                 = "vCard/ADR;HOME";
```

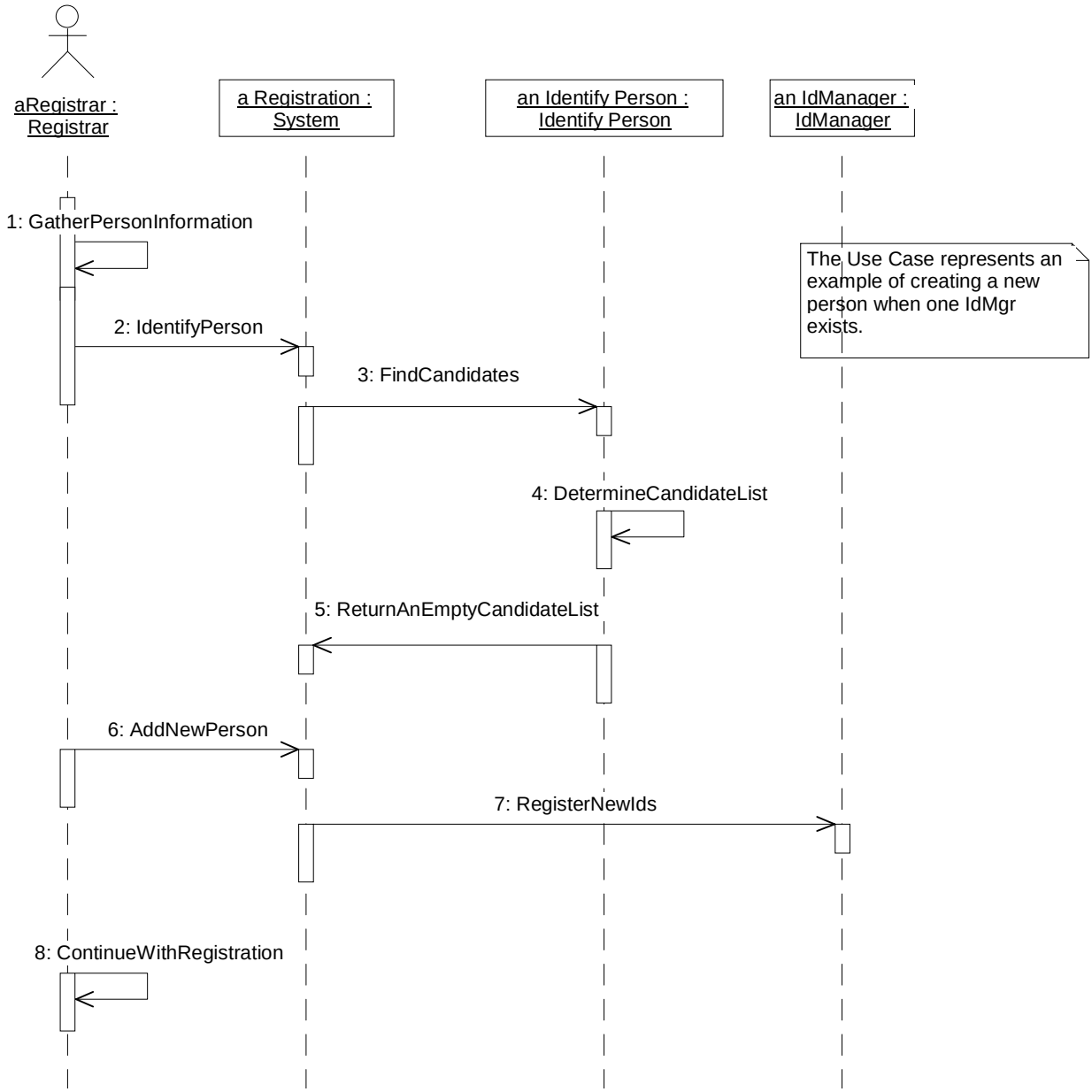
```
        const PropertyName WORK_ADDRESS                 = "vCard/ADR;WORK";
        const PropertyName TELEPHONE                    = "vCard/TEL";
        const PropertyName PREFERRED_TELEPHONE          = "vCard/TEL;PREF";
        const PropertyName HOME_TELEPHONE               = "vCard/TEL;HOME";
        const PropertyName WORK_TELEPHONE               = "vCard/TEL;WORK";
        const PropertyName VOICE_TELEPHONE              = "vCard/TEL;VOICE";
        const PropertyName FAX_TELEPHONE                = "vCard/TEL;FAX";
        const PropertyName MESSAGE_TELEPHONE            = "vCard/TEL;MSG";
        const PropertyName CELLULAR_TELEPHONE           = "vCard/TEL;CELL";
        const PropertyName BULLETIN_BOARD_TELEPHONE     = "vCard/TEL;BBS";
        const PropertyName MODEM_TELEPHONE              = "vCard/TEL;MODEM";
        const PropertyName CAR_TELEPHONE                = "vCard/TEL;CAR";
        const PropertyName ELECTRONIC_MAIL              = "vCard/EMAIL";
        const PropertyName GEOGRAPHIC_POSITION          = "vCard/GEO";
        const PropertyName TITLE                        = "vCard/TITLE";
        const PropertyName ORGANIZATION                 = "vCard/ORG";
        const PropertyName SOUND_ANNOTATION             = "vCard/SOUND";
        const PropertyName UNIFORM_RESOURCE_LOCATOR     = "vCard/URL";
};

#endif //_V_CARD_VERSION_2_1_IDL_
```

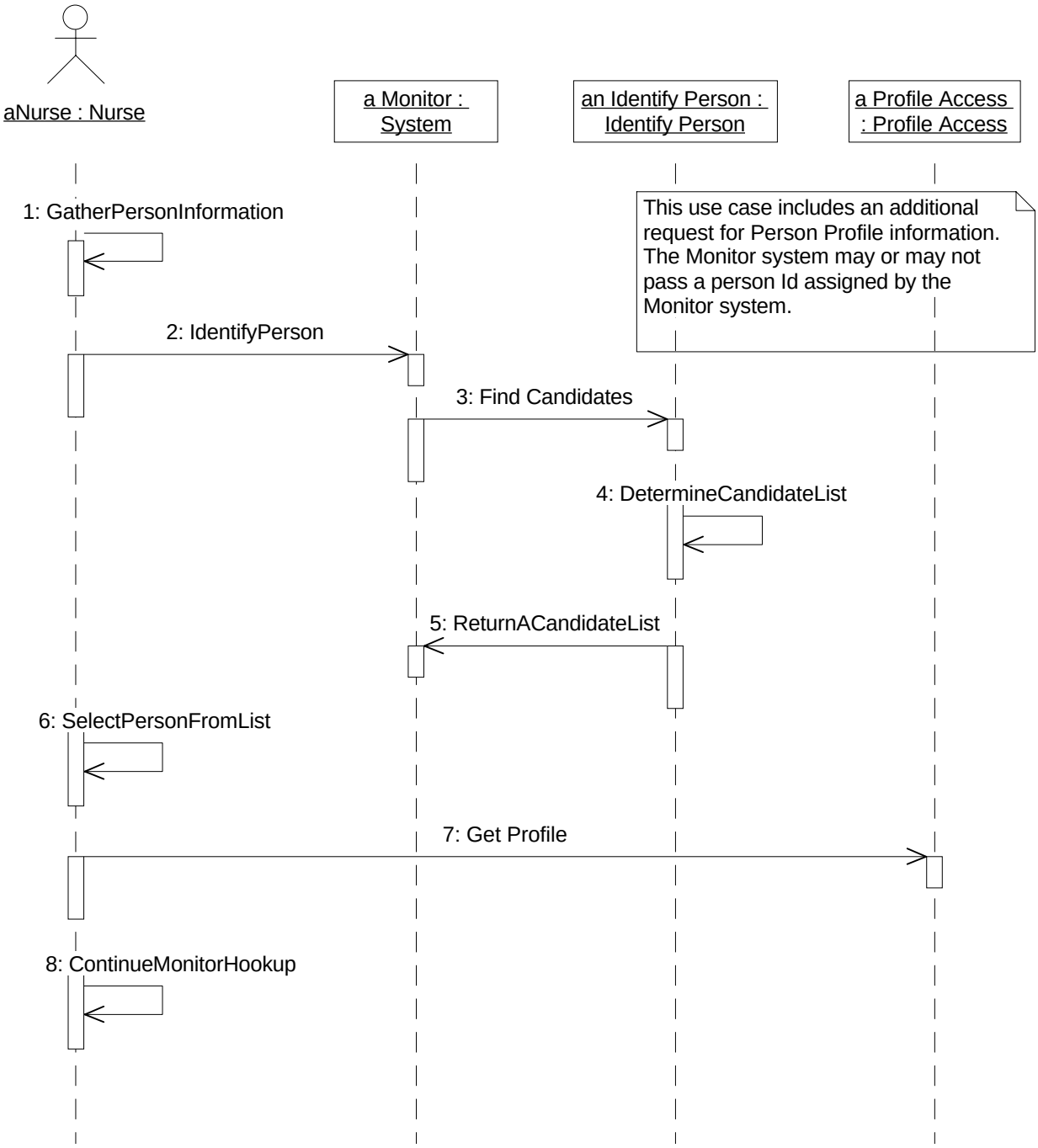
Use Case Examples *C*

C.1 Examples

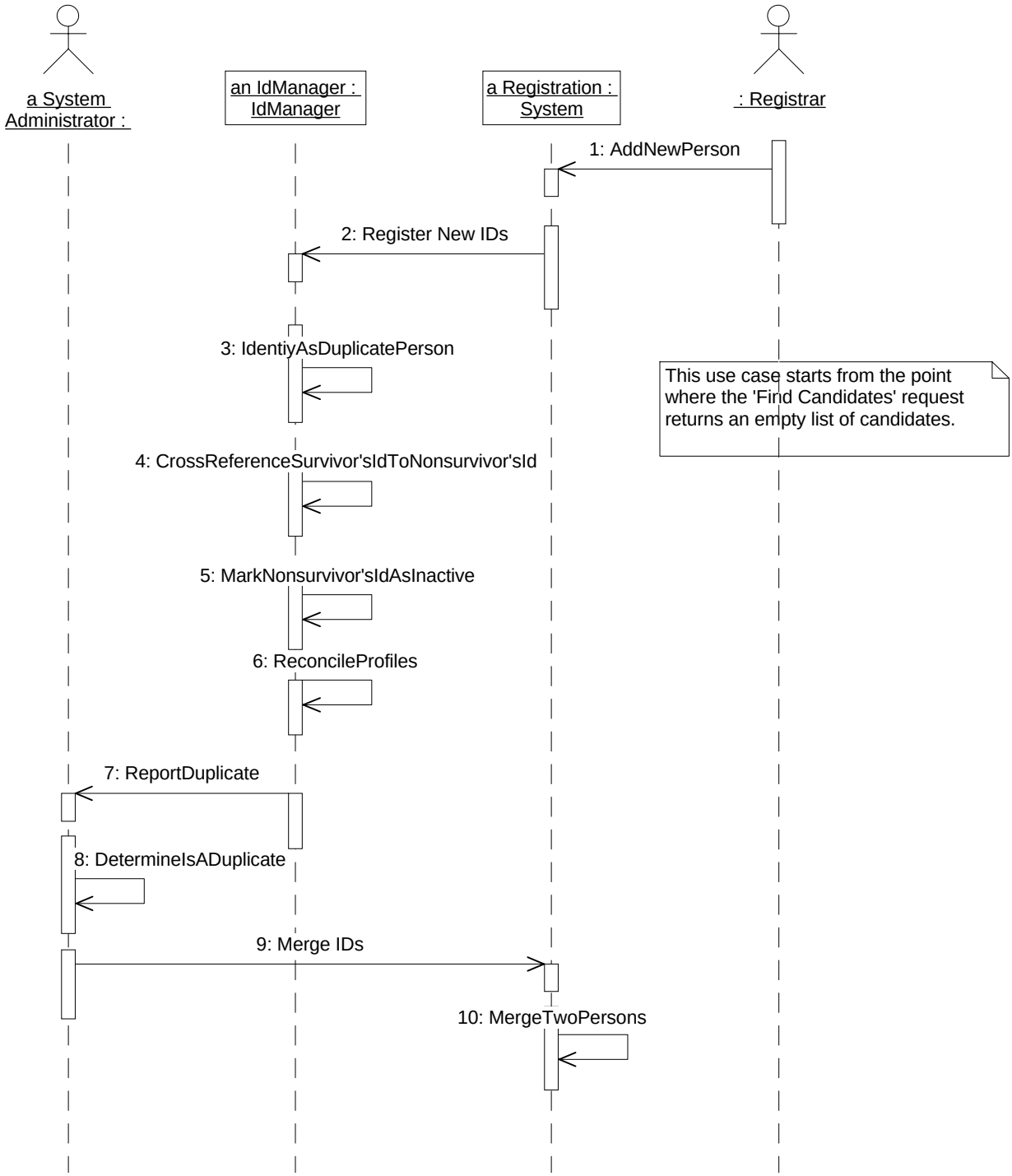
The following diagrams illustrate UseCase examples.



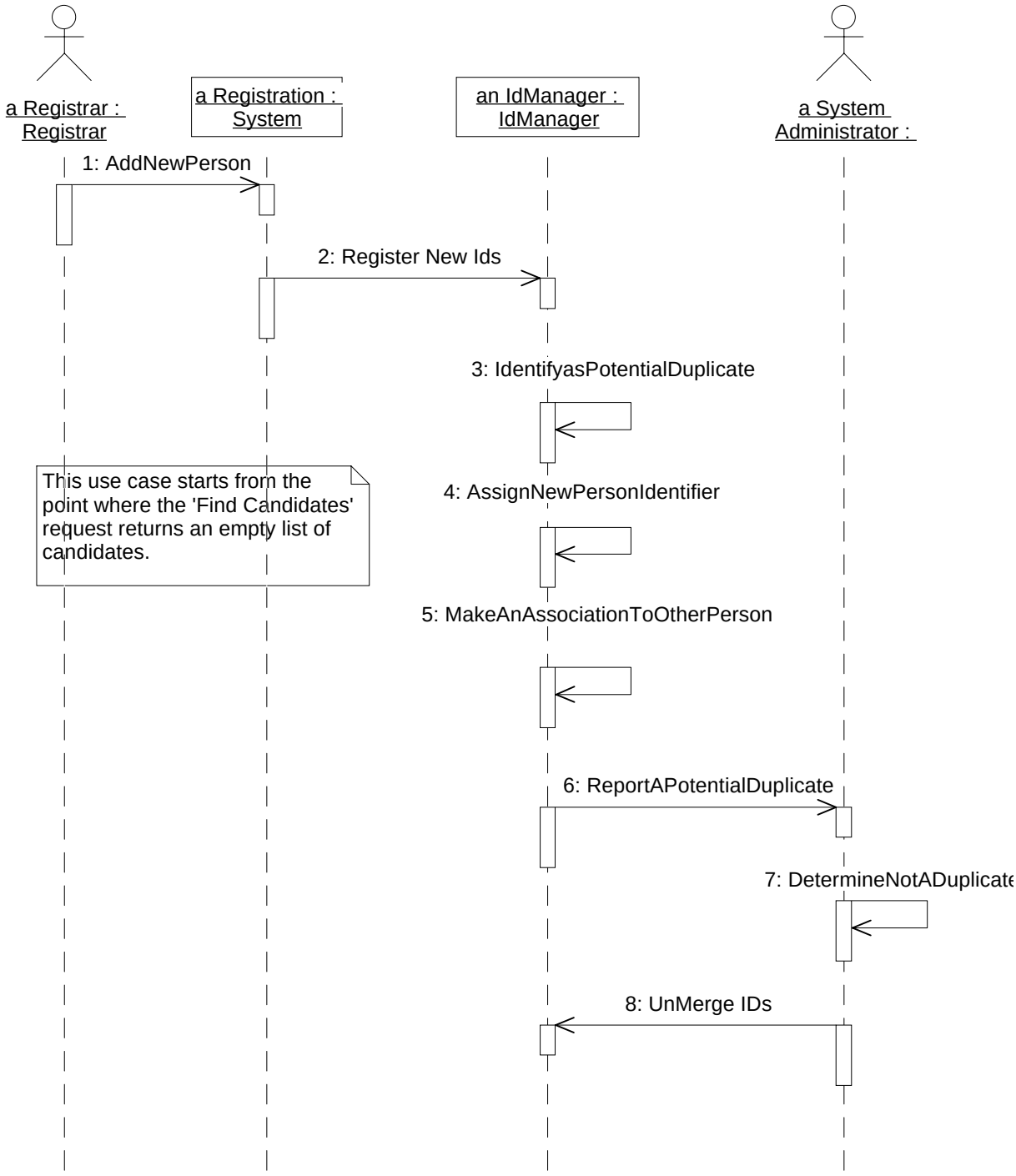
Example1 Find Candidates and Register New IDs



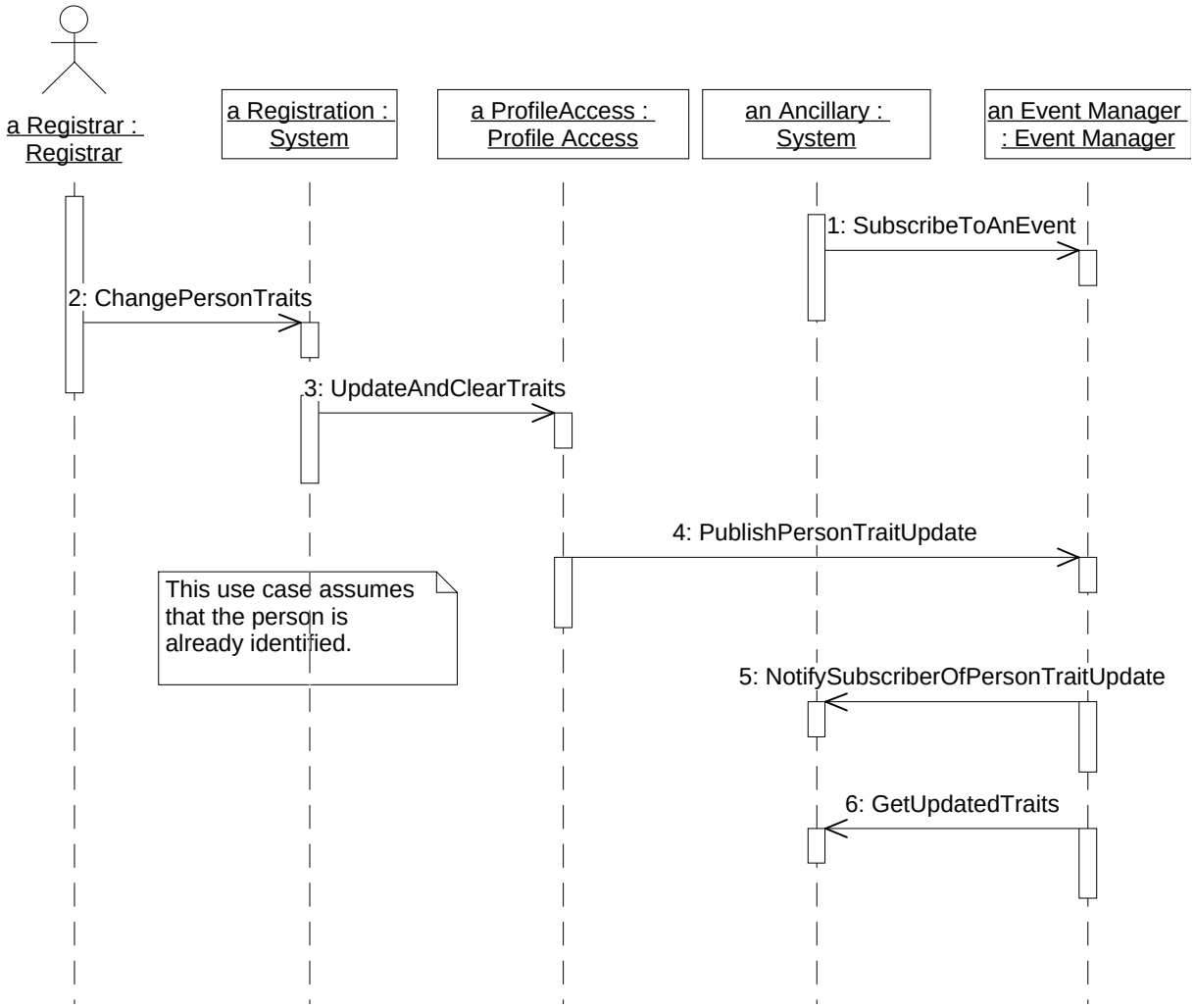
Example2 Find Candidate and Get Candidate Profile



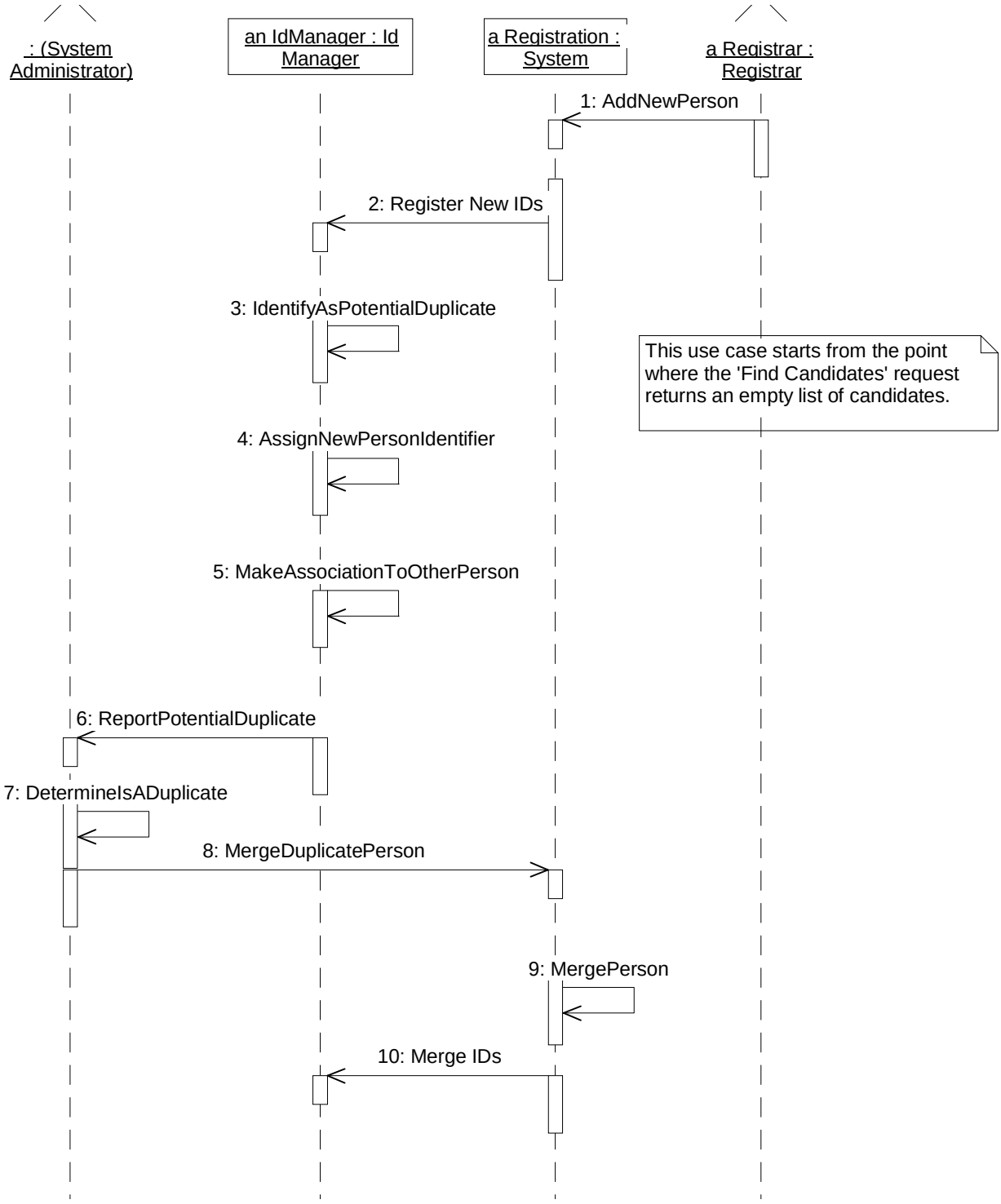
Example3 Merge IDs



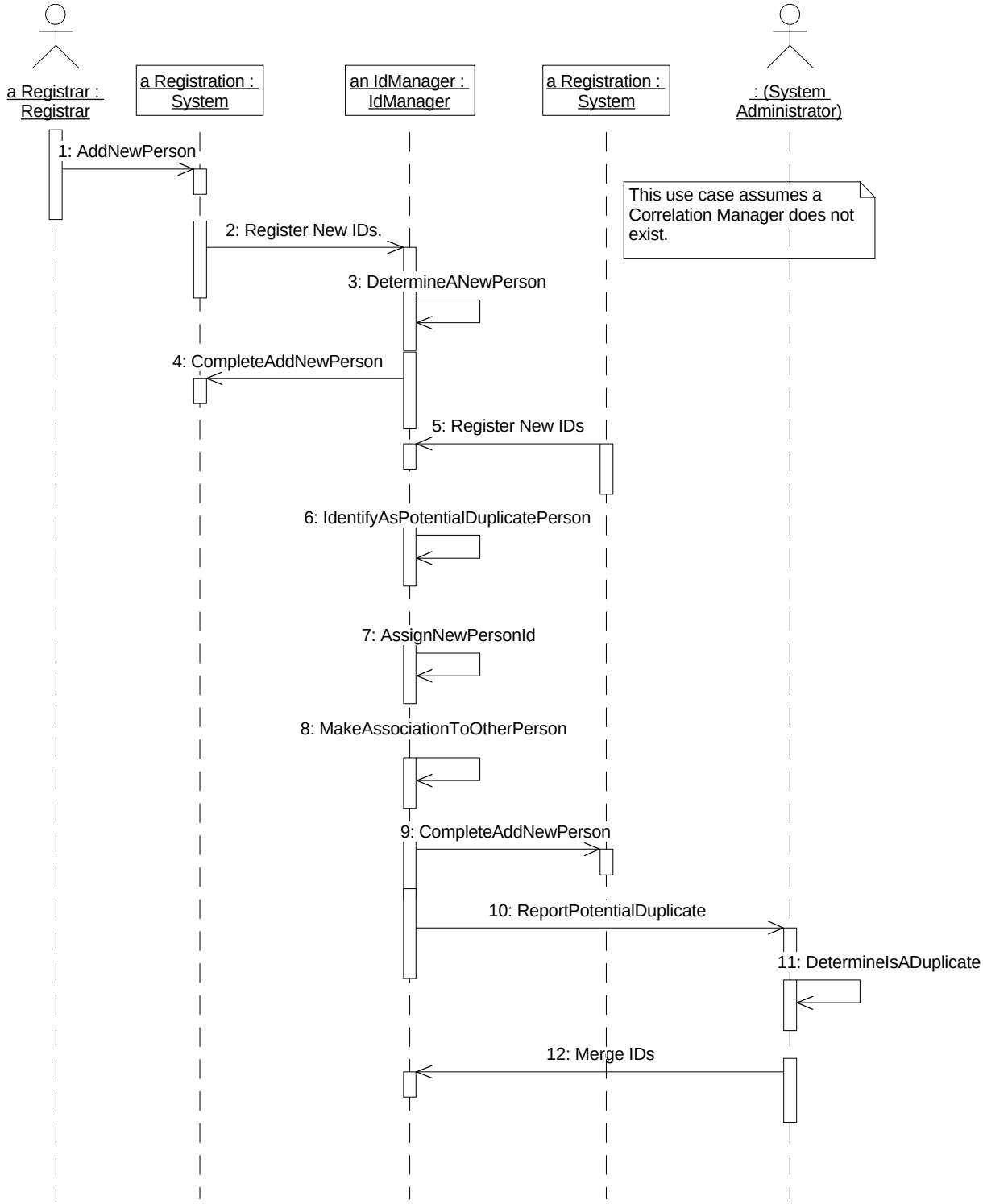
Example4 Unmerge IDs



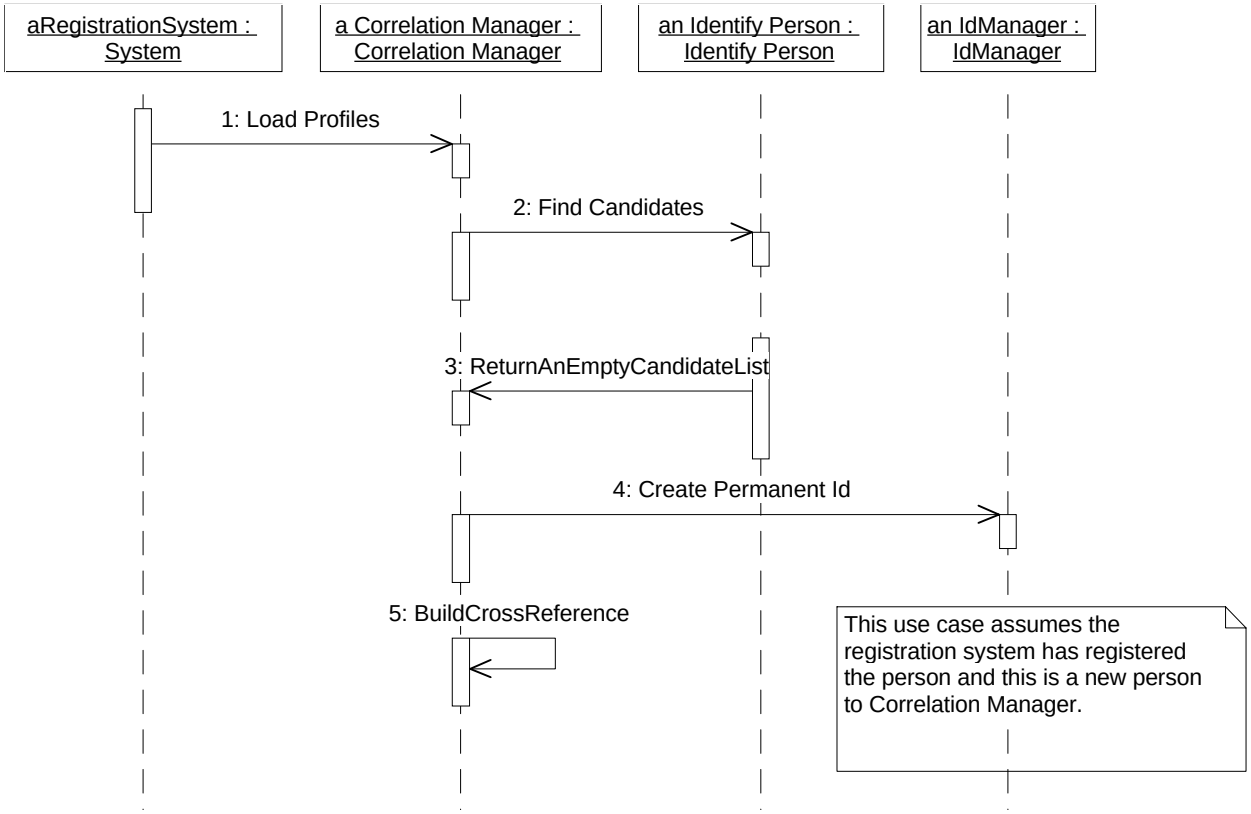
Example5 Update Person Traits and Notify Subscribed Parties



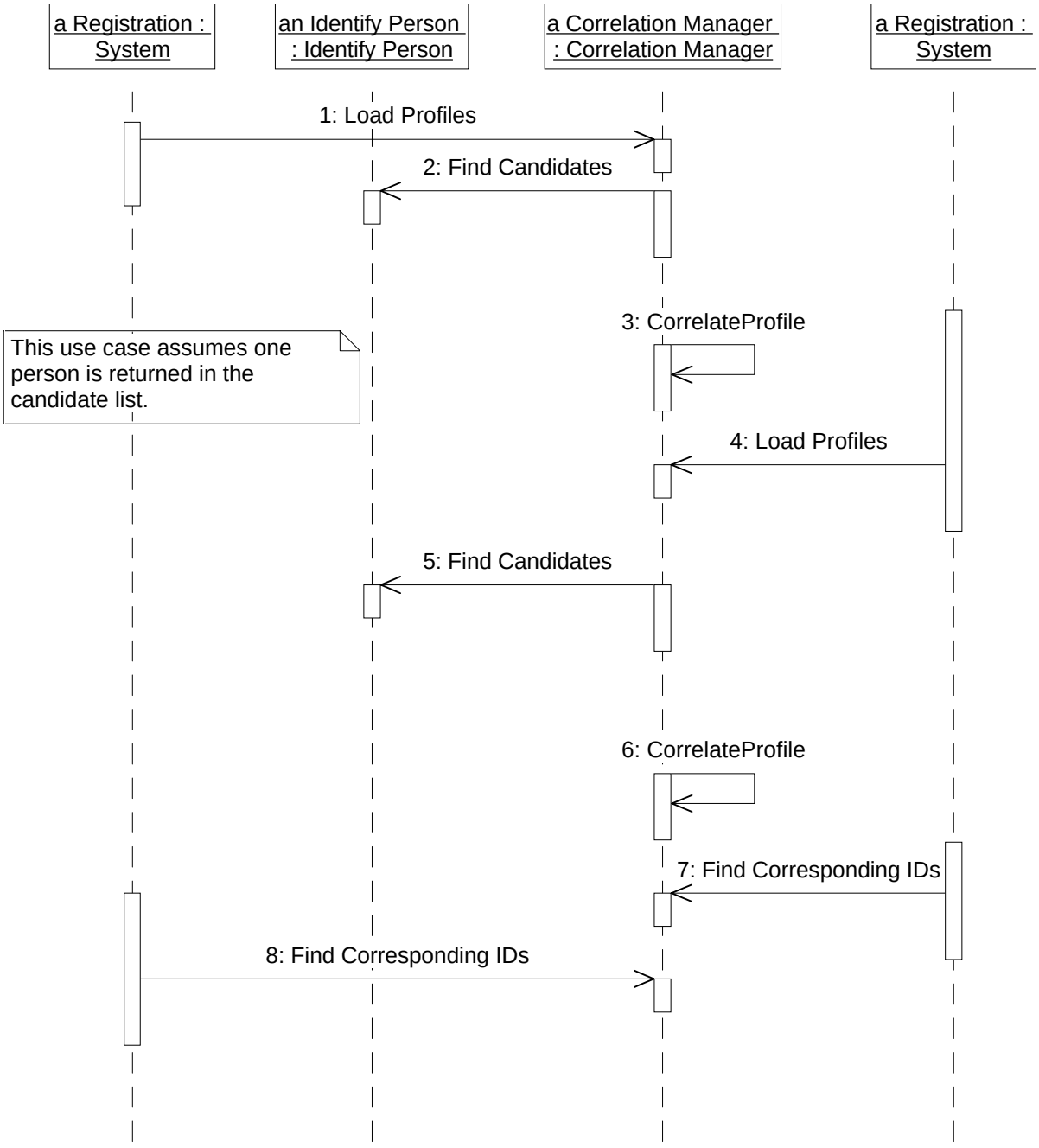
Example6 Potential Merge Person in a Single ID Domain



Example7 Merge Person without a Correlating Manager



Example8 Correlate Person Profile When Person Does Not Exist

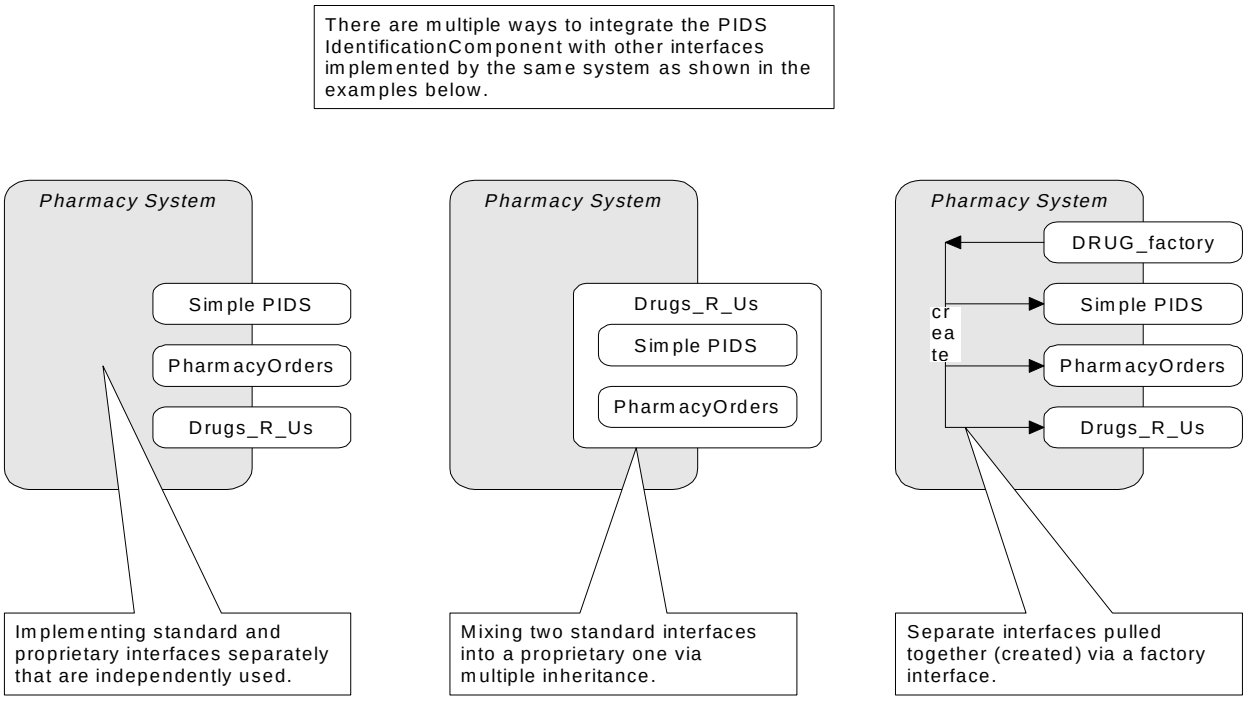


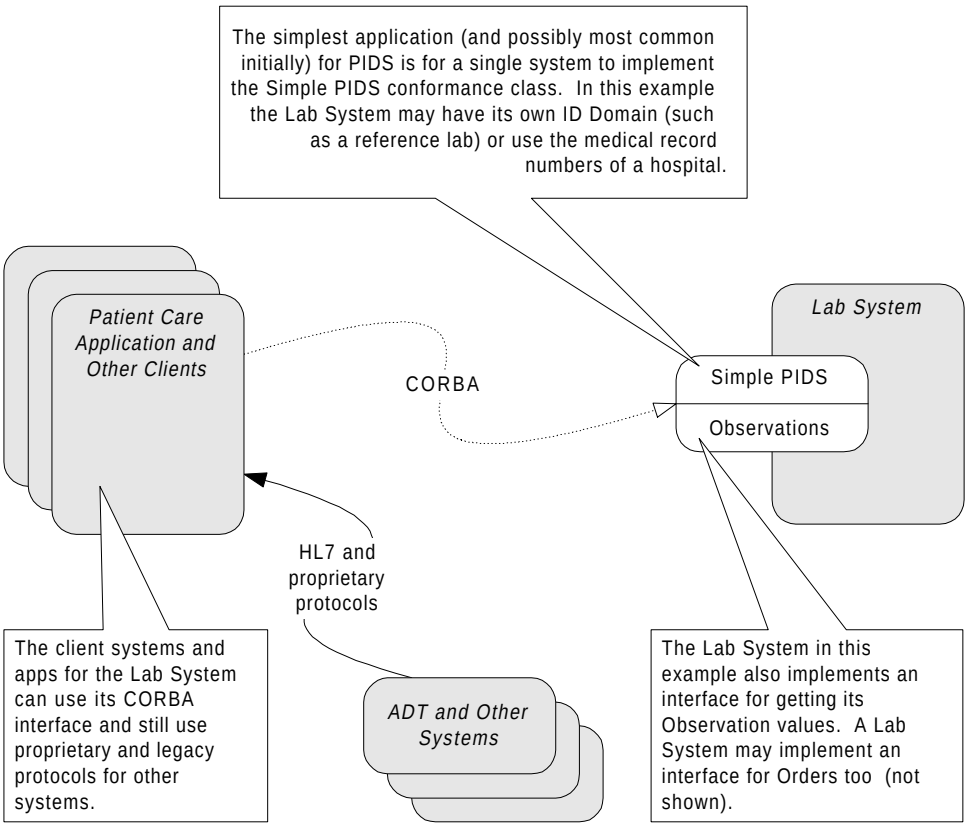
Example9 Correlate Profile When Person Already Exists

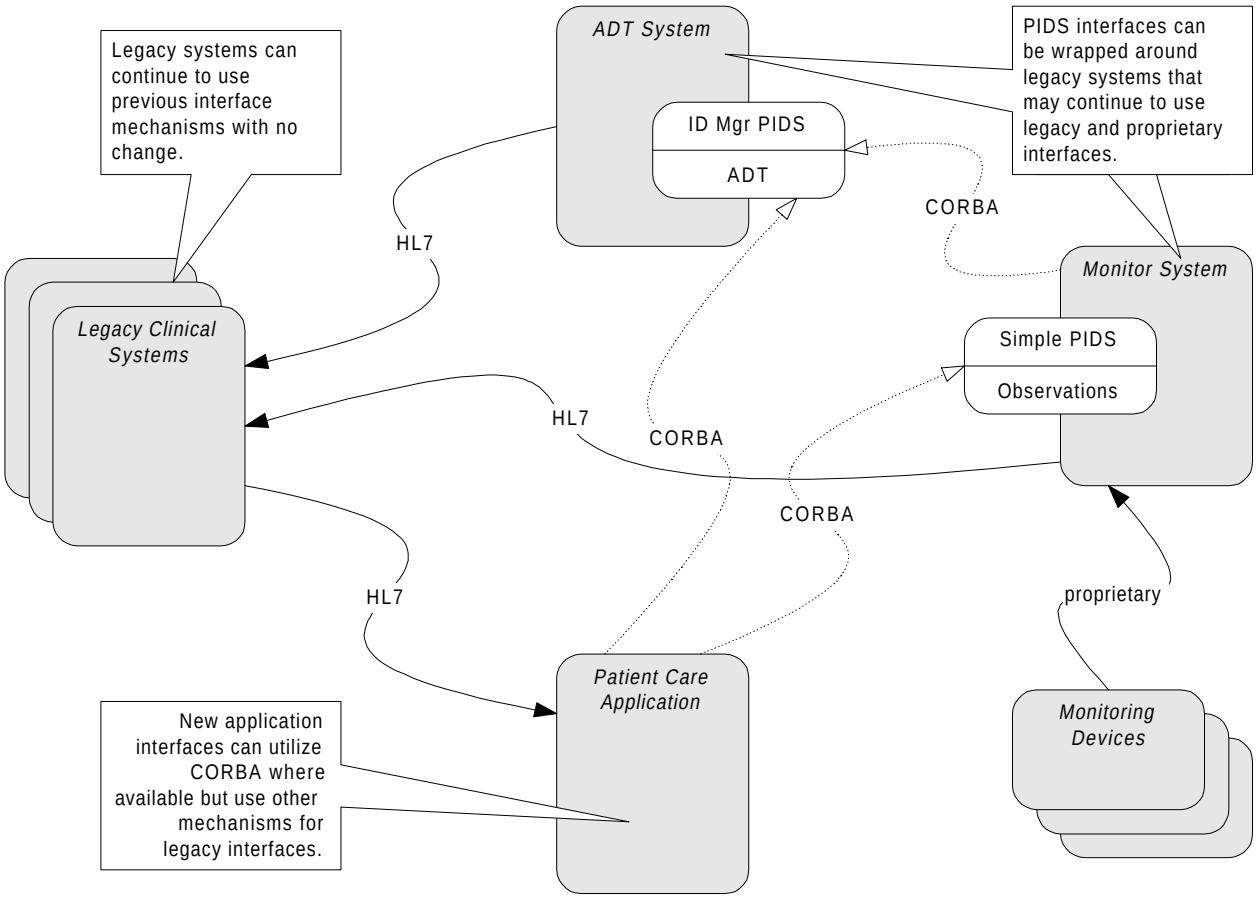
Interaction Patterns D

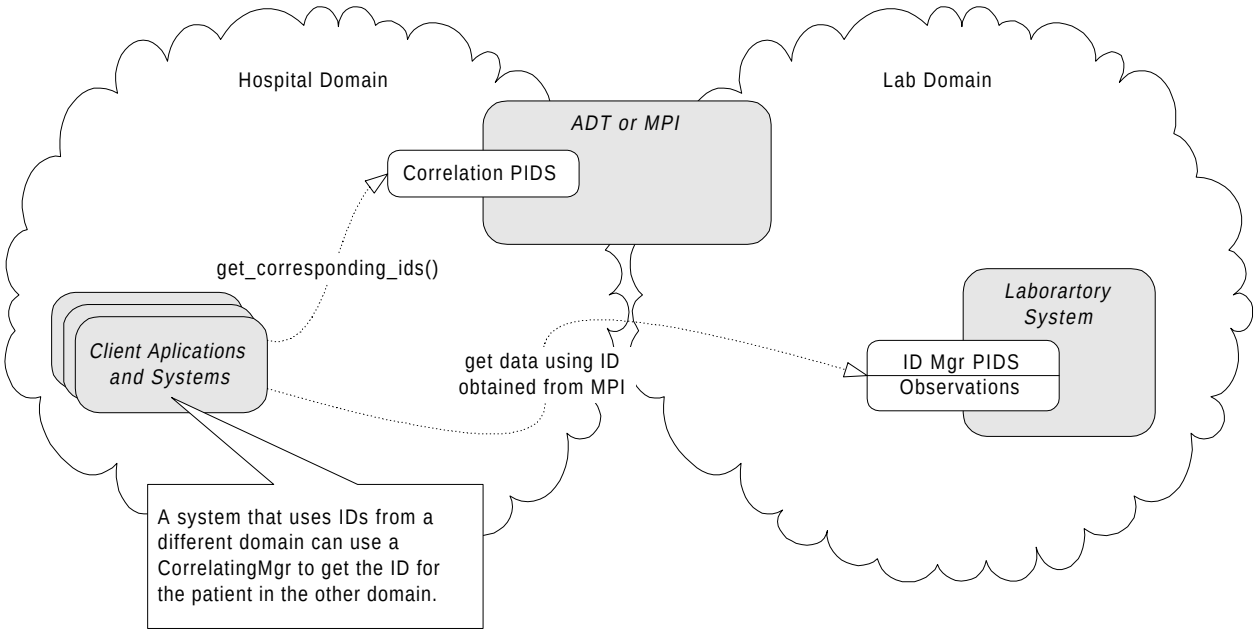
The following diagrams indicate a variety of ways that PIDS can be used. Obviously this is not a complete list, but it does show there is extreme flexibility in the way it can fit into an enterprise's architecture.

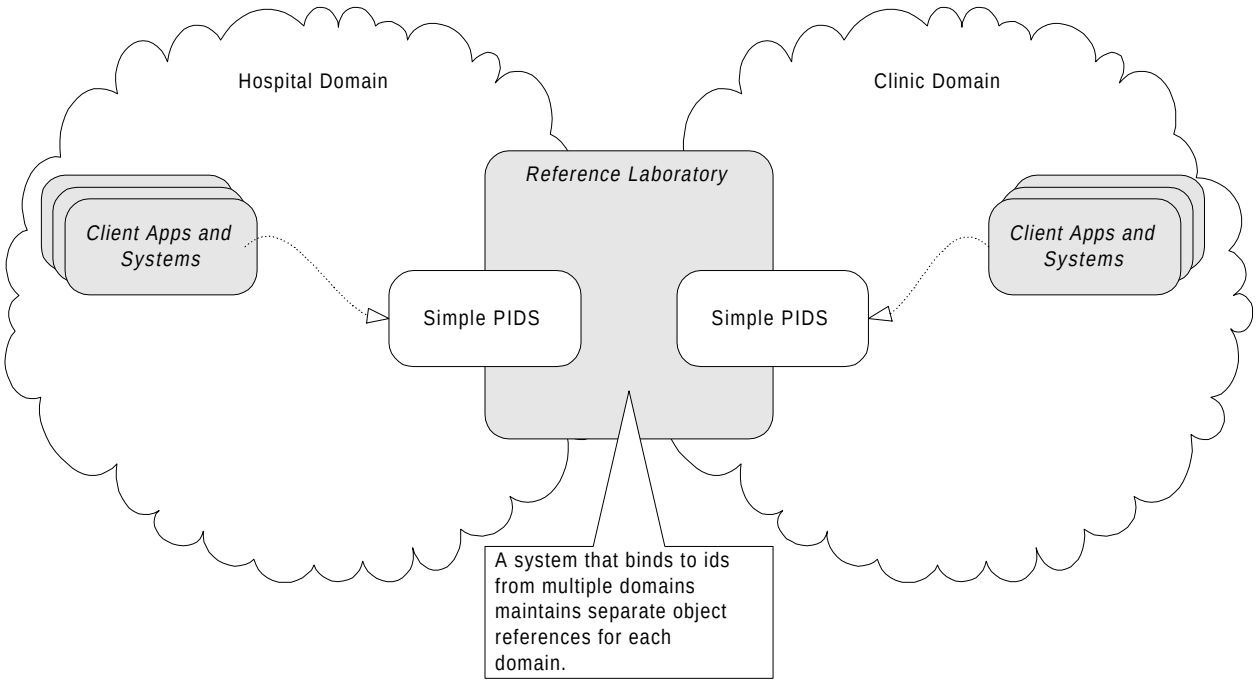
D.1 Usage Diagrams

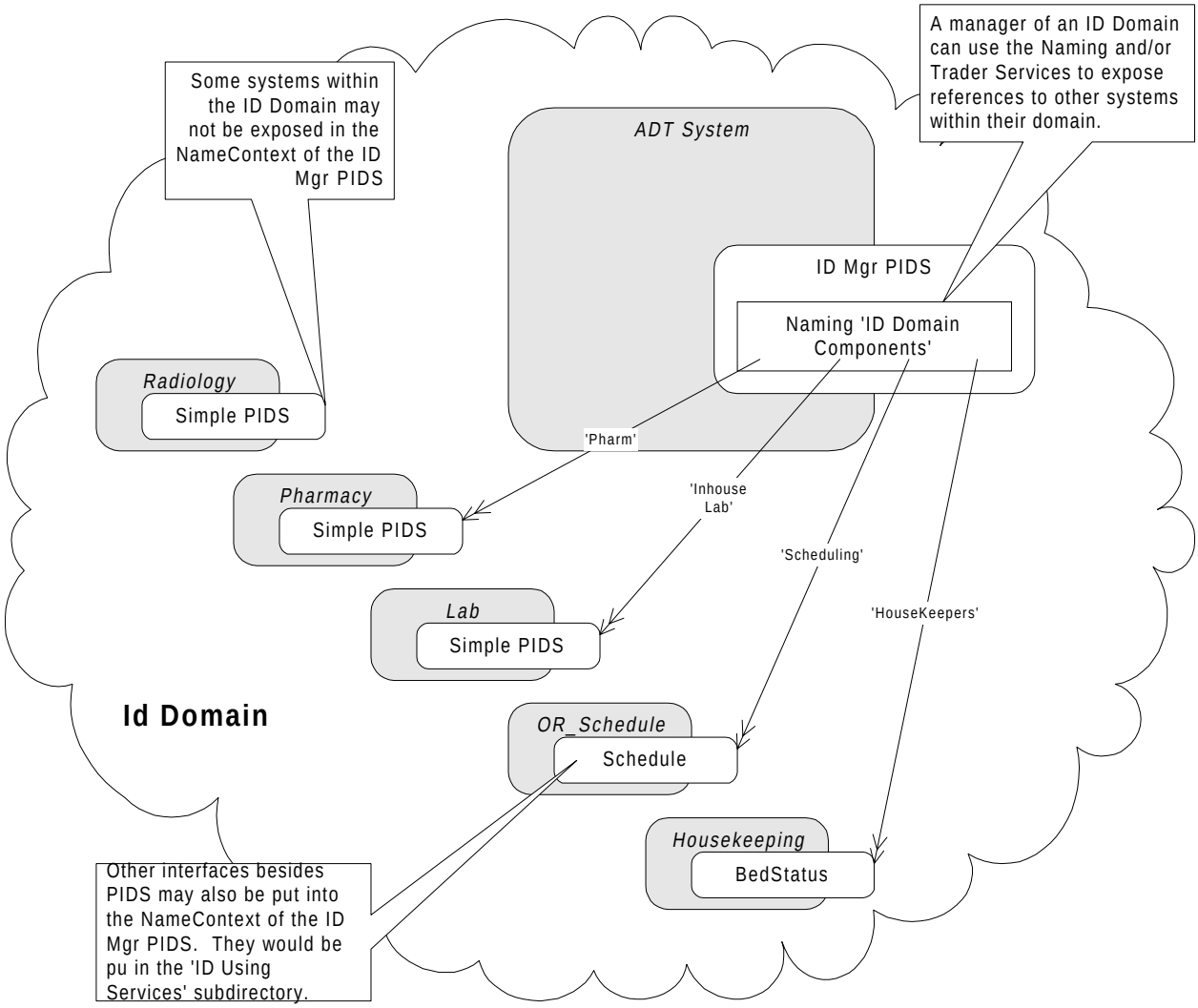


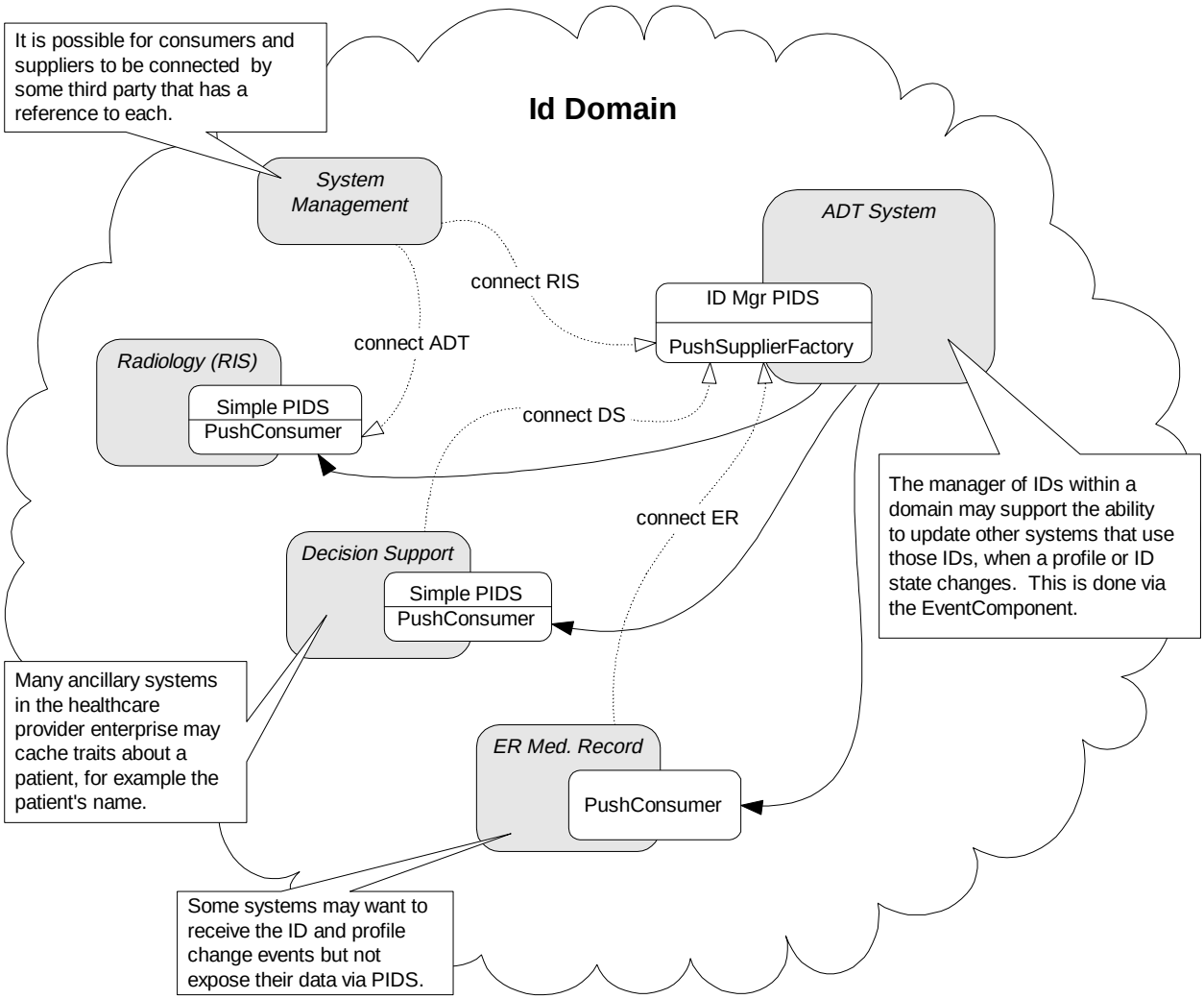

The simplest application (and possibly most common initially) for PIDS is for a single system to implement the Simple PIDS conformance class. In this example the Lab System may have its own ID Domain (such as a reference lab) or use the medical record numbers of a hospital.
Patient Care Application and Other Clients
CORBA
Lab System
Simple PIDS
Observations
HL7 and proprietary protocols
The client systems and apps for the Lab System can use its CORBA interface and still use proprietary and legacy protocols for other systems.
ADT and Other Systems
The Lab System in this example also implements an interface for getting its Observation values. A Lab System may implement an interface for Orders too (not shown).

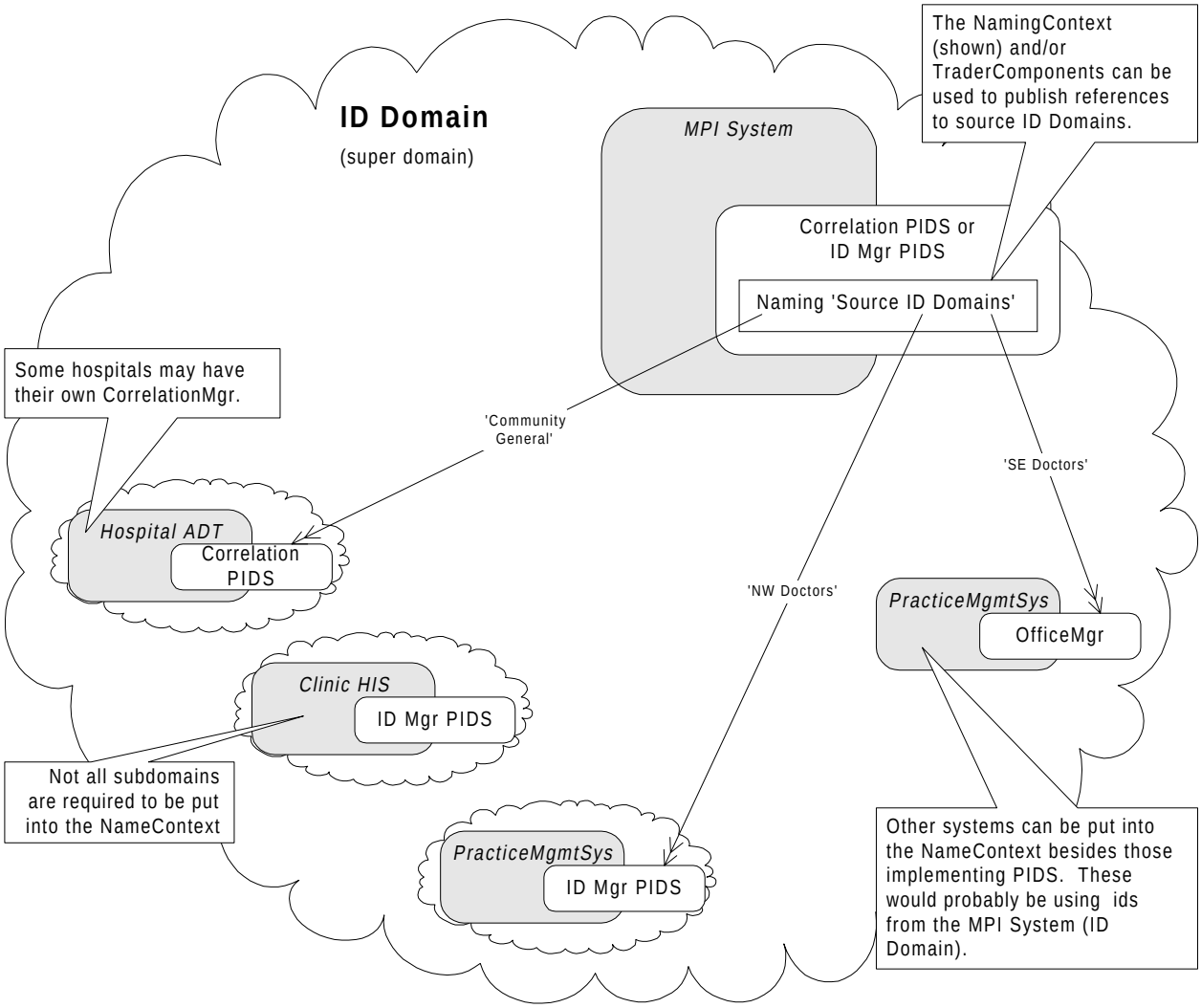

Legacy systems can continue to use previous interface mechanisms with no change.
ADT System
ID Mgr PIDS
ADT
PIDS interfaces can be wrapped around legacy systems that may continue to use legacy and proprietary interfaces.
HL7
CORBA
Monitor System
Legacy Clinical Systems
Simple PIDS
Observations
HL7
CORBA
CORBA
HL7
proprietary
Patient Care Application
Monitoring Devices
New application interfaces can utilize CORBA where available but use other mechanisms for legacy interfaces.

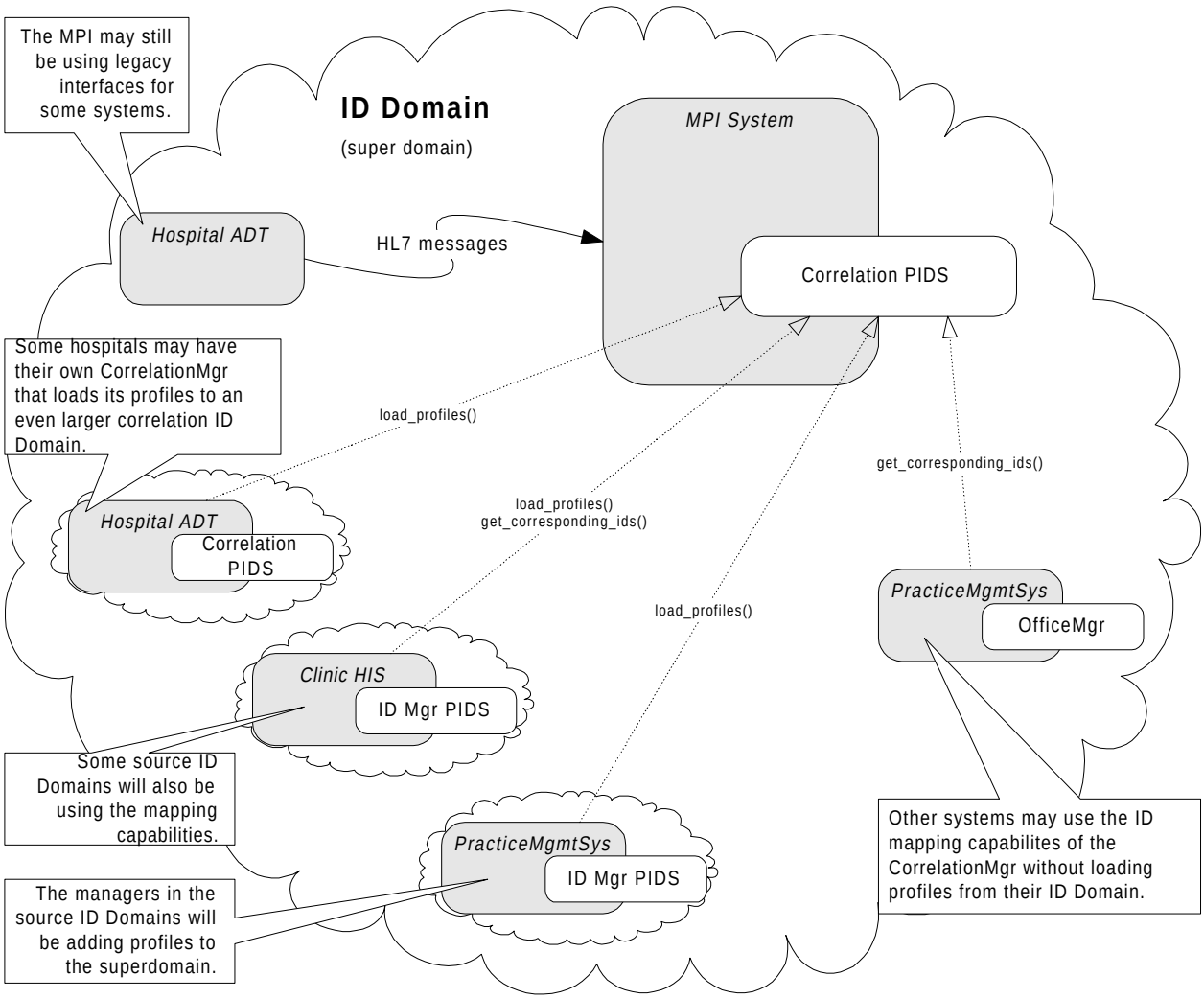

Hospital Domain
Lab Domain
ADT or MPI
Correlation PIDS
get_corresponding_ids()
Client Aplications and Systems
get data using ID obtained from MPI
Laborartory System
ID Mgr PIDS
Observations
A system that uses IDs from a different domain can use a CorrelatingMgr to get the ID for the patient in the other domain.

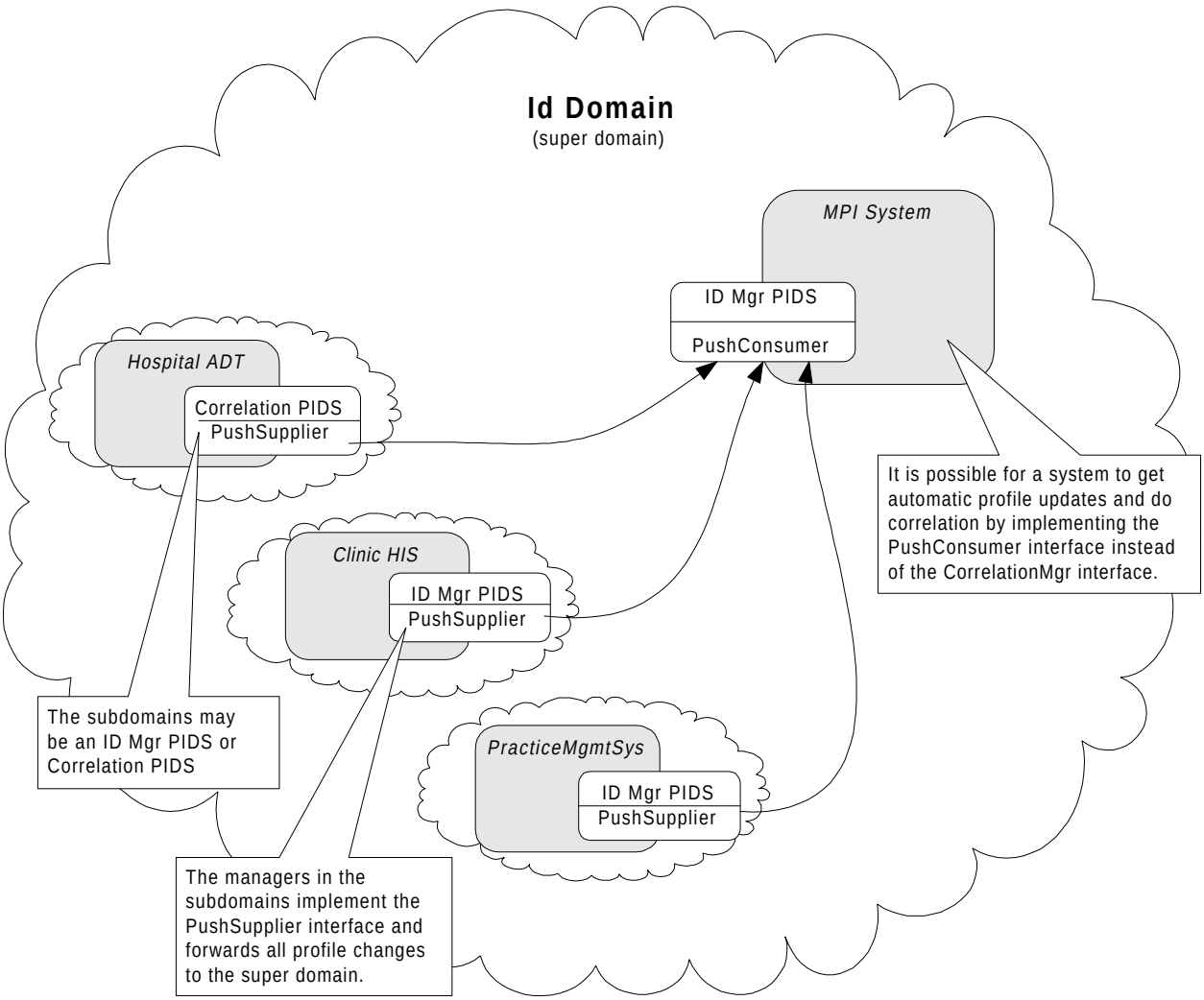

Hospital Domain
Clinic Domain
Reference Laboratory
Client Apps and Systems
Simple PIDS
Simple PIDS
Client Apps and Systems
A system that binds to ids from multiple domains maintains separate object references for each domain.

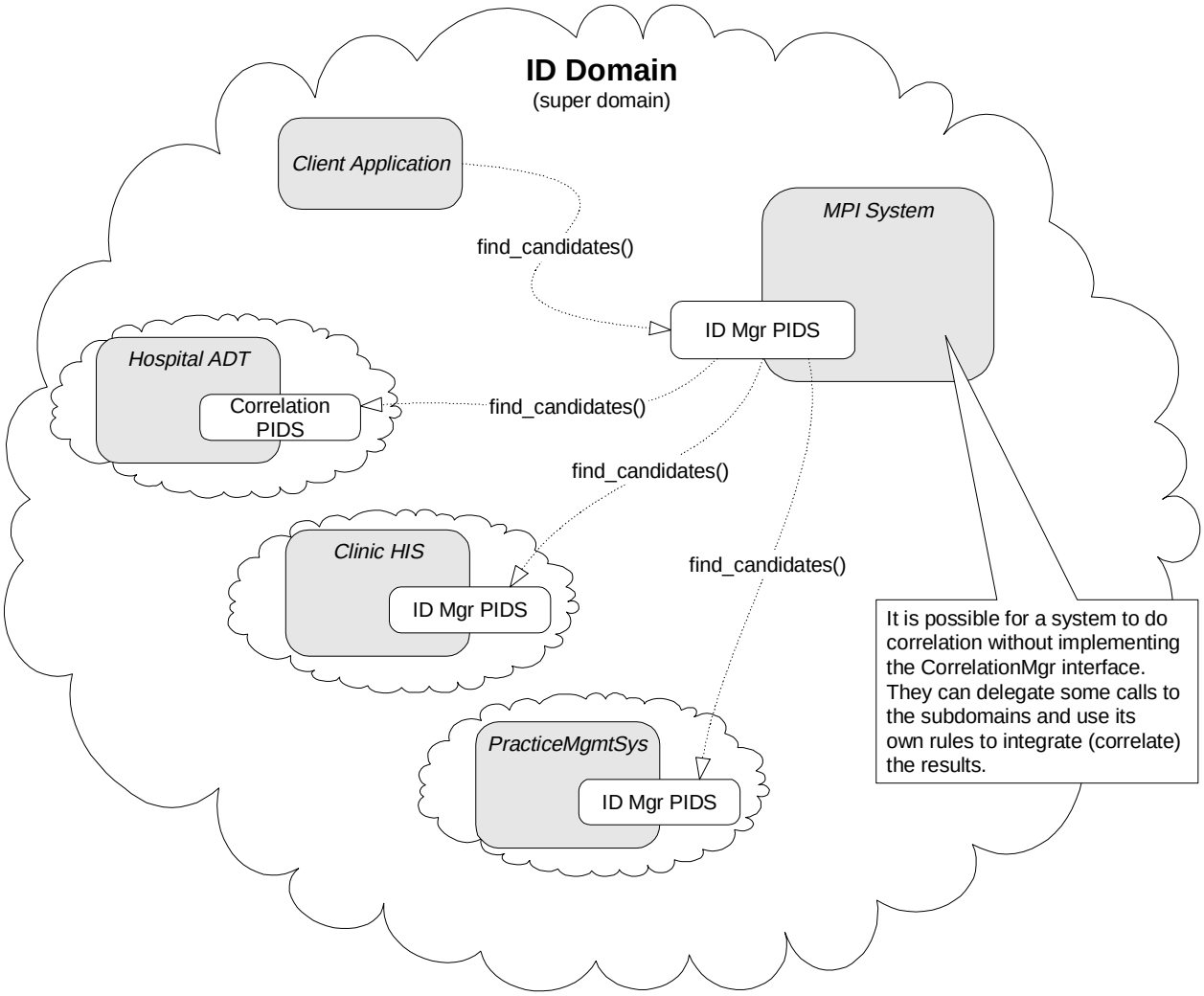

Some systems within the ID Domain may not be exposed in the NameContext of the ID Mgr PIDS
A manager of an ID Domain can use the Naming and/or Trader Services to expose references to other systems within their domain.
ADT System
ID Mgr PIDS
Naming 'ID Domain Components'
Radiology
Simple PIDS
'Pharm'
'Inhouse Lab'
Pharmacy
Simple PIDS
'Scheduling'
'HouseKeepers'
Lab
Simple PIDS
Id Domain
OR_Schedule
Schedule
Housekeeping
BedStatus
Other interfaces besides PIDS may also be put into the NameContext of the ID Mgr PIDS. They would be pu in the 'ID Using Services' subdirectory.


Id Domain
It is possible for consumers and suppliers to be connected by some third party that has a reference to each.
System Management
ADT System
connect RIS
ID Mgr PIDS
PushSupplierFactory
connect ADT
Radiology (RIS)
Simple PIDS
PushConsumer
connect DS
The manager of IDs within a domain may support the ability to update other systems that use those IDs, when a profile or ID state changes. This is done via the EventComponent.
connect ER
Decision Support
Simple PIDS
PushConsumer
Many ancillary systems in the healthcare provider enterprise may cache traits about a patient, for example the patient's name.
ER Med. Record
PushConsumer
Some systems may want to receive the ID and profile change events but not expose their data via PIDS.


ID Domain
(super domain)
MPI System
The NamingContext (shown) and/or TraderComponents can be used to publish references to source ID Domains.
Correlation PIDS or ID Mgr PIDS
Naming 'Source ID Domains'
Some hospitals may have their own CorrelationMgr.
'Community General'
'SE Doctors'
'NW Doctors'
Hospital ADT
Correlation PIDS
PracticeMgmtSys
OfficeMgr
Clinic HIS
ID Mgr PIDS
Not all subdomains are required to be put into the NameContext
PracticeMgmtSys
ID Mgr PIDS
Other systems can be put into the NameContext besides those implementing PIDS. These would probably be using ids from the MPI System (ID Domain).


The MPI may still be using legacy interfaces for some systems.
ID Domain
(super domain)
MPI System
Hospital ADT
HL7 messages
Correlation PIDS
Some hospitals may have their own CorrelationMgr that loads its profiles to an even larger correlation ID Domain.
load_profiles()
get_corresponding_ids()
Hospital ADT
Correlation PIDS
load_profiles()
get_corresponding_ids()
load_profiles()
PracticeMgmtSys
OfficeMgr
Clinic HIS
ID Mgr PIDS
Some source ID Domains will also be using the mapping capabilities.
PracticeMgmtSys
ID Mgr PIDS
Other systems may use the ID mapping capabilites of the CorrelationMgr without loading profiles from their ID Domain.
The managers in the source ID Domains will be adding profiles to the superdomain.


Id Domain
(super domain)
MPI System
ID Mgr PIDS
PushConsumer
Hospital ADT
Correlation PIDS
PushSupplier
Clinic HIS
ID Mgr PIDS
PushSupplier
PracticeMgmtSys
ID Mgr PIDS
PushSupplier
It is possible for a system to get automatic profile updates and do correlation by implementing the PushConsumer interface instead of the CorrelationMgr interface.
The subdomains may be an ID Mgr PIDS or Correlation PIDS
The managers in the subdomains implement the PushSupplier interface and forwards all profile changes to the super domain.


ID Domain
(super domain)
Client Application
find_candidates()
MPI System
ID Mgr PIDS
Hospital ADT
Correlation PIDS
find_candidates()
find_candidates()
Clinic HIS
ID Mgr PIDS
find_candidates()
PracticeMgmtSys
ID Mgr PIDS
It is possible for a system to do correlation without implementing the CorrelationMgr interface. They can delegate some calls to the subdomains and use its own rules to integrate (correlate) the results.

Event Descriptions *E*

The information in this appendix is informative and is not a normative part of the specification. It was prepared by the submitters under the expectation that there would be an adopted Notification Service.

Two of the initial submissions to the Notification Service propose the creation of event types similar to the service types defined by the Trader Service. They also propose using the constraint mechanism for event consumers to specify the filters to apply to event channels. The aspects of service types that apply to events are the type names and property descriptions as described below.

- Event Type - Event types have a name and a definition. The event may inherit from previously defined events. An inherited event has all the same properties defined by the parent plus any new type adds. The definition of the type consists of a set of properties.
- Properties - A property is a name/value pair where the value has a specific IDL type. The property is also characterized by a boolean attribute indicating if it is mandatory.

The event types and associated property descriptions needed for PIDS are described below. The syntax used is similar to the Trader service type descriptions except they do not include an interface name or have the concept of “readonly” properties.

E.1 Event Types

```
event PersonIdChange {
        mandatory property string qualified_person_id;
};

event PersonIdStateChange : PersonIdChange {
        mandatory property octet new_state;
        property octet old_state;
};
```

```
event PersonIdStateMerged : PersonIdStateChange {
        mandatory property string preferred_id;
};

event PersonIdStateUnmerged : PersonIdStateChange {
        mandatory property string old_preferred_id;
};

event PersonIdProfileChange : PersonIdChange {
        mandatory property StringSeq trait_names_changed;
        mandatory property string component_name;
        property string component_version;
        property PersonIdService::TraitSeq changed_traits;
        property PersonIdService::TraitSeq new_profile;
};

event PersonIdDuplicate {
        mandatory property StringSeq qualified_person_ids;
};

event PersonIdCollision {
        mandatory property String qualified_person_id;
};

event IdentificationComponentChange {
        mandatory property string domain_name;
        mandatory property string component_name;
};

event IdentificationComponentVersionChange : IdentificationComponentChange {
        mandatory property string new_version;
        property string old_version;
};

event IdentificationComponentTraitChange : IdentificationComponentChange {
        property StringSeq new_trait_names;
        property StringSeq old_trait_names;
        property boolean trait_spec_changed;
};

event CorrelationSourceChange : IdentificationComponentChange {
        property StringSeq new_source_domain_names;
        property StringSeq old_source_domain_names;
};
```

E.2 PersonIdChange Event

This is a general event for any changes on an ID.

qualified_person_id

This is the ID that had something change on it. It is a stringified version of the **QualifiedPersonId**. The stringification is per the rules defined in the **NamingAuthority** for **QualifiedNameStr**.

E.3 PersonIdStateChange Event

This event type is a subtype of the general **PersonIdChange** event. It applies only to state changes on the ID.

new_state

The new_state is a mandatory property which has the new **IdState** for the ID. The **IdState** enum values are coded into the octet starting with 0 and increasing sequentially in the order they are defined in the IDL.

old_state

The **old_state** is optional and represents the **IdState** value before the change.

E.4 PersonIdStateMerged Event

This event is a special case of a **PersonIdStateChange** event where the new state is Deactivated and the ID has a preferred ID set (e.g., the **merge_ids()** operation was called).

preferred_id

This is the ID that the **qualified_person_id** is merged into.

E.5 PersonIdStateUnmerged Event

This event is a special case of a **PersonIdStateChange** event where the old state is Deactivated and had a preferred ID. The ID has been reactivated (e.g., the **unmerge_ids()** operation was called).

old_prefered_id

This is the ID that the **qualified_person_id** was merged with.

E.6 PersonIdProfileChange Event

This event is a specialization of the **PersonIdChange** event where the ID profile bound to the **qualified_person_id** has changed.

trait_names_changed

This mandatory property indicates the **TraitName** for each trait that has changed in the

profile.

component_name

This property contains "**the_name**" part of the **ComponentName** where the change occurred. This may be used by a client to filter events except those from a particular component.

component_version

This property contains "**the_version**" part of the **ComponentName** where the change occurred.

changed_traits

This property contains the traits (including values) that changed. It is an optional trait since some trait values may be large (e.g., photographs) and the client will have to query for their value.

new_profile

This property contains the whole new profile. It contains a superset compared to the **changed_traits** property.

E.7 PersonIdDuplicate Event

This event indicates the PIDS has detected suspected duplicate IDs for the same person.

qualified_person_ids

These are the set of IDs the service thinks may represent the same person.

E.8 PersonIdCollision Event

This event indicates the PIDS has detected a suspected collision on an ID. That is two or more people may be represented by (or be using) the same ID.

qualified_person_id

This is the stringified **QualifiedPersonId** that a suspected collision has occurred on.

E.9 IdentificationComponentChange Event

This is a general event indicating some of the pseudo-static information on an **IdentificationComponent** has changed.

domain_name

This is the stringified ID **DomainName** in which the component resides.

component_name

This is **the_name** from the **ComponentName** on the component that changed.

E.10 IdentificationComponentVersionChange Event

This event indicates that the component's version has changed (component may have had a configuration change, software update, etc.). Whether there is any outward functionality change on the component cannot be determined for this event.

new_version

This is the new **the_version** from the **ComponentName** on the component that changed.

old_version

This is the old **the_version** from the **ComponentName** on the component that changed.

E.11 IdentificationComponentTraitChange Event

This event indicates that some aspect of the **supported_traits** attribute has changed on the component.

new_trait_names

This sequence contains the trait names for any new traits that are now supported by the component.

old_trait_names

This sequence contains the trait names for any old traits that are no longer supported by the component.

trait_spec_changed

This boolean attribute indicates that some of the values on the **TraitSpec** (mandatory, **read_only**, and/or searchable) have changed on some of the traits that were and still are supported.

E.12 CorrelationSourceChange Event

This event occurs when the source ID Domain, for which a PIDS is correlating over, changes.

new_domain_name

These are the **DomainNames** for any new source ID Domains that is being correlated over.

old_domain_name

These are the **DomainNames** for any old source ID Domains that are no longer being correlated over.

Security Guidelines F

F.1 Security Overview

The PIDS interfaces may be used in many different environments with widely varying security requirements that range from no security to extreme security. For this reason the PIDS interfaces do not expose any security information. PIDS relies on the underlying CORBA infrastructure and services which provides all the security mechanisms needed without exposing it in the interfaces.

An attribute of security that concerns many people is maintaining confidentiality of certain (sensitive) information about them. For PIDS, this implies being able to filter requests by:

- who is accessing the information,
- who the information is about,
- what information is being accessed.

Other common security concerns could be preventing unauthorized modification of data, tapping into communications to acquire sensitive information, and causing loss of service by over-burdening a service.

CORBA Security provides robust mechanisms to address these and other concerns. Some of the security properties it does deal with includes authentication, authorization, encryption, audit trails, non-repudiation, etc. CORBA Security, in its default mode, allows these security concerns to be addressed without the client and server software being aware of it. This is a powerful notion, allowing security policies to be created and enforced after applications and systems have been created and installed.

Other CORBA and CORBA Security features provide mechanisms for applications to extend these security capabilities. For example, they can obtain credentials from the ORB and implement filters that can look at specific data passed to and returned from operations.

It is a requirement of the PIDS to provide confidentiality of information that is stored about an individual. This requirement fuels the need for fine-grained access control on trait information that is associated with a PIDS ID.

PIDS provides two interfaces to access information about a person, given their ID: 1) **ProfileAccess** and 2) **IdentityAccess**. The functional capabilities of these two interfaces are identical, but they have different strengths. The **IdentityAccess** interface can simplify the internal implementation of some types of security policies - those where there is a different policy for each ID. The **ProfileAccess** interface's strength is that it only requires a single call to access data for multiple IDs. The **IdentityAccess** requires separate calls for each ID information.

The **IdentityAccess** interface allows a client to acquire an object reference to an Identity interface, containing information pertaining to only one particular ID. Creating a CORBA object reference as a single access to the information yields benefits in controlling the access to that information. It is a single point to which access of this information must flow. Since CORBA Security Services can automatically deny requests as a result of an access policy, access to information behind this interface can be controlled by that mechanism.

F.2 Security Requirements

For the PIDS to be secure in its possible dissemination of information it needs to adhere to these requirements:

- The PIDS needs to authenticate a client's principal identity, role, and sensitivity level.
- The PIDS needs to transmit information confidentially and with integrity.

The first requirement states that the entire PIDS interface implementations must be able to identify a potential client. If it cannot authenticate a client, then the client may be severely limited in the particular requests that the PIDS can service. The CORBA Security Service provides the mechanisms for a server to authenticate a client.

The second requirement provides for the confidentiality of the information. The client must communicate with the PIDS using not only encryption to protect data, but signature as well, so as not to have data tampered with during communication. There is no sense in putting a Sensitivity level of "OwnerOnly" on a trait and have its value transmitted to the owner in the clear. The CORBA Security Service provides these capabilities, including SSL.

The question is, "How does one get CORBA to support this access policy model?"

F.3 CORBA Security

In an effort to keep the PIDS interfaces security unaware (i.e., no extra visible security-relevant parameters in methods), access policy must be adhered to from behind the interfaces. The CORBA security model offers several ways to apply security policy to method invocations.

The CORBA Security Specification (CORBAsec) is not a cookbook for using CORBA security in building applications. It is a general framework with which ORB vendors and application vendors can build a multitude of different security policy models. The CORBAsec also gives the interfaces for which implementations of applications can access

those security services that are supplied with a secure ORB.

A secure PIDS implementation that can control access to specific traits must be aware of the security services offered by the ORB. This caveat also means that a client's ORB may have to know the kind of ORB and the security services that is used by the PIDS.

The CORBA security specification outlines a general security policy model. Although the specification is vague about which approach should be taken, it is specific enough to be able to choose from a couple of models that can be supported.

The CORBA security model bases itself on credentials and security domains. Credentials are data objects that contain attributes such as privileges, capabilities, and sensitivity levels, among others. Security domains are mappings from credentials to access rights. Credentials can be encrypted and signed to prevent tampering and achieve a level of trust between client and server. CORBA credentials get passed with requests beneath the visible level of the interface. CORBA security services give the clients and servers the ability to authenticate/verify credentials to implement policies in security domains.

Many different schemes, algorithms, services, and vendor implementations exist to provide implementation of security policy, and many different implementations of those schemes may be integrated into a CORBA compliant ORB. It is not the purpose of this specification to dictate the specific implementation of an ORB and security services that should be used, but to outline the external requirements for the PIDS implementation. These requirements and guidelines aid in selecting a secure ORB and the level security functionality needed to implement the PIDS access policy model.

F.4 Secure Interoperability Concerns

CORBA has built the communication bridge between distributed objects creating an interoperable environment that spans heterogeneous platforms and implementations. However, security adds another layer of complexity to the issue of interoperability. ORB implementations are not required to include security services nor provide an interoperable mechanism of security services. However, a specification does exist for the target object to advertise, via the IOR, the security services that it supports and the services it requires from the client. Both the client and server ORBs must use compatible mechanisms of the same security technology.

The CORBA Common Secure Interoperability (CSI) Specification (orbos/96-06-20) defines 3 levels of security functionality that ORBs may support. The levels are named CSI Level 0, CSI Level 1, and CSI Level 2. Each level has increasing degrees of security functionality.

The CSI Level 0 supports identity-based policies only and provides mechanisms for identity authentication and message protection with no privilege delegation. The CSI Level 1 adds unrestricted delegation. The CSI Level 2 can implement the entire CORBA Security Specification at Security Level 2.

Each CSI level is parameterized by mechanisms that can support the level of security functionality, such as SPKM for CSI Level 0, GSS Kerberos for CIS Level 0 or CIS Level 1, and CSI_ECMA for CSI Level 2. Future developments in security functionality and mechanisms are not restricted, and mechanisms can be added to each level.

The ORB implementations may use different security technology with differing capabilities and underlying mechanisms, such as SSL, DCE, Kerberos, Sesame, or other standards. Choosing the ORB and its underlying security services will be critical to protecting PIDS, and it will influence the implementation of the access policy that a secure PIDS implementation must support.

For example, an ORB that only supports SPKM (i.e., CSI Level 0) can only authenticate clients and provide confidentiality and integrity of communication. It cannot support definition and use of security attributes beyond an access ID. Support for security attributes beyond an access ID require CSI Level 2. Therefore, using an ORB that only provides CSI Level 0 will require the PIDS to maintain its own information on the credentials of clients.

Even if an ORB's security technology supports the definition of security attributes that can be delivered to the PIDS (i.e., CSI Level 2), there are still concerns involving the trust between the client and the PIDS.

F.5 Trust Models

The available trust models for the PIDS is simplistic. Since the PIDS is a communications end point and does not require requests on other services on a client's behalf, a delegation trust model is not needed. This simplifies the model and eliminates an absolute need for a CSI Level 1 or CSI Level 2 secure ORB (although they may use them).

There are two basic trust models for the PIDS. If the PIDS and its client are implemented using CSI Level 0 or CSI Level 1 ORBs, only the first trust model can be supported. If a CSI Level 2 ORB is used, both trust models can be supported. The trust models are:

1. The client's identity can and is trusted to be authenticated. However, the client is unable or not trusted to deliver the valid credentials.

2. The client is trusted to deliver the correct credentials.

In the first model, the client ORB is required to authenticate its principal (the user) and provide authentication information to the server ORB. The methods used to accomplish principal authentication is specific to the mechanisms (e.g., DCE or Kerberos) that the selected ORB supports. Management of those identities is also specific to the mechanism. The server ORB must have a compatible mechanism that verifies the authentication information and carries out mutual authentication of the client.

With this trust model, a secure PIDS implementation must maintain and manage a map of identities to privilege attributes. CSI Level 0, 1, and 2 ORBs are able to support this trust model.

Even if the ORB has CSI Level 2 functionality, it may be a local policy that a PIDS does not trust the credentials brought forth from an authenticated client. In that case, the PIDS must maintain the map or use a default set of security attributes for requests from clients it does not trust.

In the second model, the client ORB is required to authenticate its principal and acquire its valid credentials. The methods used to accomplish principal authentication and acquisition of privilege attributes are specific to the mechanism that the selected ORB supports, such

as DCE and Sesame. Management of those identities and attributes are also mechanism-specific. A secure PIDS installation using this trust model must take a careful look at that management scheme and operation, evaluate it, and decide to trust it. In such a scenario, the server ORB, which has CSI Level 2 functionality, automatically verifies the credentials on invocation.

A secure PIDS built to the second model leaves management of identities and their attributes to the security services policy management system used by the ORB. The PIDS may manage security attributes for the data itself.

A secure PIDS built to the first model will have some scheme to manage trusted identities and their credentials. There is no interface or plan in the PIDS to specify this kind of management.

F.6 CORBA Credentials

To adhere to the credential model that supports trait-specific access policy, a set of credentials must contain privilege attributes such as the identity of the client, the role in which the client is actively represented, and the sensitivity level of information to which the client is allowed access. It will be the responsibility of a PIDS implementation to advertise to potential client vendors the specifics of these attributes and how to represent them externally. A client ORB needs to ascertain certain credentials about the user and must pass them to the PIDS. An external representation of those credentials is needed so that credentials can be passed between client and server within the CORBA security services. The CORBA Security module defines the structure for this representation.

```
module Security {

const SecurityAttributeType  AccessId  = 2;
const SecurityAttributeType  Role      = 5;
const SecurityAttributeType  Clearance = 7;

struct SecAttribute {
    AttributeType    attribute_type;
    Opaque           defining_authority;
    Opaque           value;
};
typedef sequence<SecAttribute> AttributeList;
}
```

Listed above are the relevant pieces of the specification from the **Security** module that apply to externalizing credential information.

The **Security::AccessId** security attribute type could represent the person for which the ID and hence Identity object reference relates to. In constructing the value of a **Security::SecAttribute** of this type, the defining authority part could be the name of the PIDS ID Domain manager, and the value part could be the ID within that domain. However, if the ORB uses an underlying scheme where the value of the AccessId security attribute is supplied by some security services, such as a DCE name, a map to the PIDS ID may be needed.

The **Security::Role** security attribute type should represent the mandatory role. The

defining authority part could take the name of the PIDS ID Domain that specifies the role, and the value can be the ID within that domain.

The **Security::Clearance** security attribute type can be used to represent the Sensitivity Level. For example, the values could be represented by the strings, "OwnerOnly," "LevelA," "LevelB," "LevelC," "None," and "Undefined."

F.7 CORBA Security Domain Access Policy

In addition to a credential based scheme, CORBA defines security domains. The purpose of this section is to explain and illustrate the use of the standard CORBA security policy domain and the way in which it may be used to implement a security policy for the PIDS. This section offers a recommendation to a PIDS implementor to give a feel for the kinds of security policy a PIDS implementation may need to support. It should also guide the implementor in evaluating a secure ORB and available security services.

A security domain governs security (access) policy for objects that are managed within that domain. In order to make scalable administration of security policy, these domains map sets of security credentials to certain sets of rights. A right is a sort of an internal security credential.

CORBA defines a standard set of rights that are granted to principals within a security domain. A security domain administrator manages that map through the **SecurityAdministration** module's **DomainAccessPolicy** interface. Access decisions can then be based on a set of required rights and the rights granted to the client by the domain access policy, by virtue of the client's credentials.

ORB security service vendors will supply a security policy management infrastructure that implements the standard CORBA rights scheme. The PIDS must use security services that can place different required rights on the PIDS interfaces. Some ORB security services may allow a security domain to create special rights. However, CORBA defines a standard set of rights: get, set, and manage. This right set will suffice to handle the PIDS.

In the model just described there is one security domain for all of the PIDS components. The CORBA rights families scheme within a single security policy domain suffices to differentiate the security nature of the methods. More generally any number of domain models can be used, such as a separate security domain for each PIDS component.

The PIDS interfaces are divided up so that for most of the interfaces one right can apply to all methods of each interface. The following table recommends the required rights for each of the PIDS interfaces. An asterisk implies that the listed right applies to all methods in the interface that are not listed separately.

Interface	**Required Rights**
PersonIdService module	
IdentificationComponent::*	corba:g
ProfileAccess::*	corba:g
ProfileAccess::update_and_clear_traits()	corba:s
IdentityAccess::*	corba:g
IdentityAccess:: update_and_clear_traits()	corba:s

SequentialAccess::*	corba:g
IdentifyPerson::*	corba:g
IdMgr::*	corba:s
CorrelationMgr::*	corba:g
CorrelationMgr::load_profiles()	corba:s
Notification module	
Filter::*	corba:s
EventComponent::*	corba:g
Push/PullSupplier/ConsumerFactory::*	corba:m
Push/PullSupplier/Consumer::*	corba:s

Most methods on the **IdentificationComponent**, its subtypes, and the EventComponent can be considered "get" methods. The domain access policy for the security domain should grant authenticated clients with the proper access credentials (i.e., access ID and role) with the get (corba:g) right.

All the methods in the **IdMgr**, **Filter**, and consumer/supplier interfaces as well as the **update_and_clear_traits()** and **load_profiles()** operations can be considered "set" methods. These “set” methods change information; therefore, they have a different security function other than the other methods. A client that is granted the right "get" should not necessarily be allowed access to methods that can change information. Clients that are allowed to change information in the PIDS should be granted the set (corba:s) right.

The **Factory** interfaces perform management of event end-points; therefore, it is recommended that access to these objects should be more limited. The manage (corba:m) right may be sufficient to separate this duty from the others in a single security domain.

F.8 Request Content-Based Policy

The CORBA standard domain access policy scheme only protects methods from invocation at the target and without regard to content of the request. The PIDS needs a more fine grained access control in order to implement the content-based access policy required (e.g., access policies for different traits). The PIDS implementations must be made security-aware to implement an access policy based on the value of arguments in a request. There are multiple ways to implement this policy using a secure CORBA implementation.

The CORBA Security Specification supplies two different schemes represented by an interface hierarchy, which are Security Level 1 and Security Level 2 (these should not be confused with CSI Levels 0, 1, and 2). These interfaces describe the level of security functionality that is available to security-aware implementations.

Security Level 1

For the PIDS to take advantage of CORBA security in order to implement its access policy model, the ORB must at least implement the CORBA Security Level 1 interfaces. A

Security Level 1 compliant ORB supplies an interface to access the attributes of the credentials received from the client.

Using the **SecurityLevel1** interfaces, which is simplistic, enables implementation of the PIDS interfaces to examine the client's credentials and compare them to the access control information that is managed as the access policy; however, the implementation of the PIDS must be security-aware.

```
module SecurityLevel1 {

Current get_current();

interface Current {
    Security::AttributeList get_attributes(
            in  Security::AttributeTypeList attributes
    );
};
}
```

Using the Security Level 1 interfaces, each implementation of a PIDS query interface must call the **get_attributes()** function on the **Current** pseudo object, examine the attributes, compare to the access policy information, and make the access decision. The implementation should raise an exception if access is determined to be denied.

It is the responsibility of the client's ORB to acquire the proper credentials securely. It is the responsibility of the server's ORB to authenticate credentials received from the client, extract the security attributes from them, and make them available to the implementation through the **Current::get_attributes()** method.

Security Level 2

Using an ORB which supplies the Security Level 2 interfaces, the implementation can be somewhat free of making the access control decision in the implementation of the query interfaces. Having an implementation that is security-unaware is attractive in CORBA, because security policy decisions can be made underneath the functionality, and they have the ability to be changed without retooling the application.

As with any framework, there are several ways in which to use the Security Level 2 interfaces. One approach could be to implement a replaceable security service for the access decision. Security Level 2 describes a method in which security can be enforced by the creation of an Access Decision object. The **AccessDecision** object would interact with a **DomainAccessPolicy** object to get effective rights and compare those to rights returned from the **RequiredRights** interface.

Some secure ORB implementations may allow the installation of specialized Access Decision objects to be used in conjunction with specialized **DomainAccessPolicy** objects. In the Security Level 2 interfaces, the specification implies access control only on the invocation of a method and not the contents of the request.

```
module SecurityReplaceable {

interface AccessDecision {
```

```
    boolean access_allowed (
        in  SecuirytLvel2::CredentialList  red_list,
        in  CORBA::Object                  target,
        in  CORBA::Identifier              operation_name,
        in  CORBA::Identifier              interface_name
    );
};
}
```

Currently, the **AccessDecision** object specified in the **SecurityReplaceable** module does not take the invocation **Request** as an argument. It only makes an access decision based on the credentials received from the client, the target object reference and operation name, and the target's interface name. This criteria is insufficient to implement the content-based access policy, if needed by a PIDS implementation to be automatically performed by the ORB.

Since the PIDS requires access control on the contents of the method invocation (such as asking for the value of the **HomePhone** trait), this scheme of replacing these Security Level 2 components cannot be used. ORB security services that use the standard CORBA domain access policy may use third-party implementations for these components. This standard domain access policy functionality gives the PIDS a high level of invocation protection that is orthogonal to the content-based access policy. Some PIDS need the standard domain access policy functionality in addition to providing content-based access policy; therefore, another approach must be taken.

A content-based access policy can be implemented in a Security Level 2 ORB by using an interceptor. A request level interceptor takes the **Request** as an argument; therefore, it can examine the content of the invocation arguments.

```
module CORBA {

interface Interceptor { ... };
interface RequestLevelInterceptor : Interceptor {
    void client_invoke( inout Request request );
    void target_invoke( inout Request request );
};
}
```

Installing an interceptor on an ORB is ORB implementation specific, and each ORB vendor may have its own flavor of interceptors. The ORB calls the request level interceptor just before the invocation gets passed to the server implementation by using the **target_invoke()** operation. The interceptor uses the Dynamic Skeleton Interface (DSI) to examine values of the arguments of the invocation and make access decisions. These access decisions are also based on the credentials received from the client and the access policy. The interceptor will deny access to the invocation by raising an exception. The server's ORB will transmit this exception back to the client.

The use of the interceptor scheme frees the implementation of the PIDS interfaces from the implementation of the access decision policy. If the access policy model changes, then the interceptor can be changed without retooling the PIDS implementation.

As awareness of the need for more powerful and flexible security policy management grows, more tools to create, manage, and analyze policy will come into existence. A PIDS

implementation relying on interceptors to implement its security policy may be able, with relative ease, to switch to using more robust policy services as they become developed.

Glossary

Terminology

Ancillary System

A subordinate, secondary, or auxiliary system within an organization. An ancillary system could have its own simple PIDS and/or be sending requests to other PIDS. Examples: Vital Signs Monitoring Systems, Laboratory Information Systems (LIS), Scheduling Systems, Pharmacy Systems, Radiology Information Systems (RIS), etc.

Attended Matching

Matching - Matching that occurs as a result of human interaction. See *Matching*.

Bind

To logically attach or associate information about a person to an ID. Within an ID Domain, it can be said that an ID is a shorthand representation of a real person or a key to more information about a person. For example, if a person is admitted to a hospital for the first time, a registration clerk enters identifying information into a system. All or part of this information is used to build a profile for the person. The system assigns an ID for the person and associates, or binds the profile to the ID.

Candidate

A person returned in the matching process that meets matching criteria. For example, if a clerk enters identifying information about a person and searches a PIDS system for a match, the PIDS system returns one or more candidates along with indications of how well each candidate matches the entered information.

Client	Any system or application that accesses or requests service from a Person Identification Service.
Collision	Within an ID Domain, a situation in which an ID is suspected to have been used by more than one person. Contrast with Duplicate.
Component	A cohesive set of software services. In this specification, a PIDS implementation is referred to as a component. A PIDS component implements varying PIDS interfaces as defined by PIDS conformance levels. For example, a Simple PIDS supports IdentifyPerson and ProfileAccess interfaces.
Confidence or Confidence Level	A matching algorithm's measure of probability that a candidate is a match. When a matching process returns a list of candidates it also returns a confidence level value with each entry in the list of candidates. The range of values for the confidence indicator is 0.0-1.0 with 1.0 being the higher confidence (e.g., 100%).
Correlating ID Domain	An ID Domain that correlates one or more other ID Domains. For example, a PIDS in a healthcare setting can be set up to correlate IDs from multiple providers (hospitals, clinics, Physicians offices, etc.) and multiple ancillary systems (lab, pharmacy, registration, etc.), where each participating system implements a different ID Domain.
Correlation	The creation of a cross-reference or mapping between Person IDs within a single ID Domain or across multiple ID Domains. For example, a PIDS set up to correlate IDs from a hospital and a lab stores both the hospital's ID and the lab's ID for any person with an ID in both ID Domains.
CPR	Computerized Patient Record
Deprecate	To indicate that an ID is not valid any more, within an ID Domain.
Deactivate	To deactivate an ID. No new information may be recorded under a deactivated ID, but it is allowed and accepted that some already exists. For example, when merging duplicate IDs in a PIDS, one of the IDs remains active. The other is deactivated, or marked as inactive. See also: deprecate. Contrast with Collision.
Domain	See *ID Domain*.

Domain Name

The name of an ID Domain in which an ID has meaning. That is, IDs are only relevant in a particular ID Domain. Each ID Domain has a Domain Name that is unique and different from all other ID Domain Names.

Duplicate or Duplicate ID

An ID is deemed to be a duplicate when it refers to the same person as one or more other IDs within an ID Domain. This results in the person "being in the ID Domain more than once." IDs that are known to be duplicates should be merged such that the associated person has one and only one unique ID and profile in the corresponding ID Domain.

Federation

As it relates to PIDS federation, is the ability to structure ID Domains into hierarchies where the higher-level ID Domains contain IDs for a superset of the persons with IDs in the lower-level ID Domains. Operations such as searches for persons can be performed the same on all levels but applies only to the IDs known at that level and below. PIDS implementations can manage each of these ID Domains in which case the PIDS is said to be federated. PIDS provides a **CorrelationMgr**, **NamingContext**, **TraderComponents**, and **EventComponent** specifically to facilitate federation.

ID or Identifier

A sequence of characters that one or more systems in an ID Domain use to represent a person and bind related information. This could be numeric, alpha, and may include punctuation, etc.

Identity

The distinct real-world person an ID and profile represents. In a PIDS, an ID is established that represents a person's identity where each ID corresponds with one real-world person. As an IDL interface, 'Identity' instances correspond one for one with a particular real world person that has been represented by an ID in the ID Domain.

Identification

The process of assigning an ID or finding an ID based on knowing some traits about the person.

ID Domain or Domain

A set of person IDs among which there is to be one unique person ID value per person or entity represented. For example, a hospital Admission, Discharge & Transfer computer system creates IDs for people as they are entered into the system. The set of IDs it manages is an ID Domain. ID Domains have a Domain Name which uniquely identifies it from other ID Domains.

People can have an ID from many ID Domains. Therefore, a person ID value has meaning for identification only if the correct ID Domain qualifies the ID value. For example, in the USA, the ID value 123-45-6789 can be used to identify a person if it is prefaced with the Social Security Number acronym, SSN, and it was assigned by the Social Security Administration.

Multiple systems can 'reside' in an ID Domain if they utilize/reference person IDs from the same ID Domain. For example, a lab system and a billing system can use the same medical record numbers to identify people. Each system can be said to 'reside' in the same ID Domain.

MPI and EMPI

Master Patient Index and Enterprise Master Patient Index.

Matching

The process that determines from a set of traits whether a person may already be known to a PIDS. The matching operation may return zero to many persons depending on the algorithm, weights on traits, and threshold parameters used in the matching process.

Merge

To apply an operation on two or more IDs representing the same person in an ID Domain, which then results in one active ID for that person. All except one of the IDs are deactivated. In other words, Merge operations are used to rectify the discovery of Duplicate ID's. Contrast with Correlation, where IDs are not deactivated.

Naming Authority

Any organization that assigns names determines the scope of uniqueness of the names and takes the responsibility for making sure the names are unique within its name space. In the same way that ID values are meaningful only within the context of their ID Domains, names are unique only within the context of their naming authority.

PersonID or Person ID

Same as ID or Identifier.

PIDS	The term PIDS is used in two ways: 1) to represent this Person Identification Service specification; and 2) to represent conforming implementations to this specification.
Profile	A set of information about a person that can be used to identify him/her. A profile consists of one or more Traits.
Unmerge	To take a person ID that has been merged with one or more other IDs and undo the merge, resulting in two or more person IDs. The person profiles are bound back to their original IDs before the merge. For example, suppose Person A has ID 1 and Person B with ID 2 had been merged into ID 1. A successful unmerge operation would restore Person A with ID 1 and Person B with ID 2.
Subdomain	If an ID Domain is being correlated over by a Correlating ID Domain, then it can be called a Subdomain of the Correlating ID Domain.
System	An application or set of applications that interact with each other, interact with the PIDS or implement PIDS. System in this context is synonymous with application. Examples of systems might include a hospital or clinical information system, an ancillary system such as a lab or radiology system, or a financial/administrative system such as an ADT.
Trait	An attribute (i.e., information) that can be used to help identify a person. Traits are grouped to create a profile. Examples of a trait include name, date of birth, sex, address, etc.
Unattended Matching	A matching process that occurs without human intervention. See *Matching*. For example, an automated process may be configured to run once a night to scan an ID Domain, searching for potential duplicate person entries. Also when profiles are added to an ID Domain and the PIDS automatically determines if an ID already exists for the person.

Index

Index

Index