Digital Persona: A Gateway to Personal Information

by

Sabra-Anne R. Kelin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999

© Massachusetts Institute of Technology 1999. All rights reserved.

Author .................................................................

Department of Electrical Engineering and Computer Science

February 3, 1999

Certified by ..............................................................

Robert C. Barrett

Research Staff Member, IBM Almaden Research Center

Thesis Supervisor

Certified by ..............................................................

Daniel Jackson

Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by ..............................................................

Arthur C. Smith

Chairman, Department Committee on Graduate Students
Digital Persona: A Gateway to Personal Information

by

Sabra-Anne R. Kelin

Submitted to the Department of Electrical Engineering and Computer Science
on February 3, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A Digital Persona (DP) is a gateway to a set of personal information, such as phone numbers and email addresses. Applications and devices which use personal information, such as electronic calendars and cellular phones, can manage this information by using a DP. Information which is accessible through a DP can be stored in the DP itself or in other applications. Since a DP facilitates the transfer of information between devices and applications, a DP can help solve many information management problems, such as handling data with different internal representations. The DP's structured, yet flexible, object model allows diverse devices and applications to access and update information without being forced to adopt a common object model. This thesis discusses the design of the DP object model and API, as well as the DP's implementation. Implementation details which are covered include object-oriented techniques, reflection, and interfaces.

Thesis Supervisor: Robert C. Barrett
Title: Research Staff Member, IBM Almaden Research Center

Thesis Supervisor: Daniel Jackson
Title: Associate Professor of Electrical Engineering and Computer Science
Acknowledgments

Completing this thesis would not have been possible without the help, kindness, and patience of many people. First and foremost, I would like to thank Rob Barrett, my supervisor at IBM. As those who have worked with Rob know, he is a wonderful advisor, teacher, and friend. I will always be grateful to Rob for inspiring me to challenge myself and encouraging me when I was doubtful of my abilities. When I first embarked on the project, I did not see all of the implications of my work. Gradually, through many discussions and design sessions with Rob, I began to see the bigger picture and appreciate his grand vision of what a Digital Persona could be. By helping me with everything from abstract ideas to implementation details, Rob played a key role in making the Digital Persona a reality.

Many thanks also go to the members of the USER Group, especially Jörg Meyer, Steve Ihde, Paul Maglio, and Mike Wirth. During my time at IBM, Jörg and Steve were great officemates and friends. I could always count on them to answer questions and offer advice on how to proceed with my project. Paul and Mike gave me valuable insights involving the object model and the use of reflection and meta information.

I would also like to thank Daniel Jackson, my MIT advisor. Our discussions helped me solidify my understanding of the project and its place in a larger context. Thanks also to IBM and the MIT VI-A program for giving me the opportunity to work at Almaden as an undergraduate and for sponsoring me during my graduate semester.

On a more personal note, I would like to thank Derek Truesdale for being such a terrific friend over the past two years. It would have been much more difficult to finish this thesis, let alone MIT, without him.

Finally, I would like to thank my parents for all of the support which they have given me over the years. Without their guidance, I would not have achieved nearly as much.
To My Parents
# Contents

1 Introduction ................................ 11
   1.1 The Problem With Personal Information Applications .......................... 11
   1.2 The Solution ................................................. 12
      1.2.1 Overview ................................................. 12
      1.2.2 Designing the Solution: Features of the System ...................... 15
   1.3 The Information Object Model ................................ 20
      1.3.1 Structure and Flexibility .................................. 20
      1.3.2 Progressive Revelation: The Best of Both Worlds ................... 22
      1.3.3 DP Programming Style ..................................... 24
   1.4 External Management of Information ................................ 28
   1.5 Thesis Overview .............................................. 28

2 DP Internals ............................................. 31
   2.1 Object Type System ............................................. 31
      2.1.1 Overview ................................................. 31
      2.1.2 Kinds, Interfaces, and Implementations .......................... 33
   2.2 Structure Surrounding Objects .................................. 42
      2.2.1 Motivation ................................................. 42
      2.2.2 Interfaces and Containment ................................ 43
      2.2.3 The DP Itself: The Outermost Container Object .................... 44
   2.3 DPObject: The Building Block of a DP ................................ 46
      2.3.1 DPObject and the Class Hierarchy for Implementations ............ 46
      2.3.2 Methods Defined in DPObject .................................. 48
2.3.3 Why DPObj ect Works (Java and Reflection) ............... 53
2.3.4 How DPObj ect Is Used In Progressive Revelation ....... 55
2.4 Summary .................................................................. 57

3 DP Ontology .................................................................. 59

3.1 The Need For a Default Ontology ............................... 59
  3.1.1 A Starting Point .................................................. 59
3.2 Our Ontology and its Representation ............................ 60
  3.2.1 Our Representation .............................................. 61
  3.2.2 Features of Our Ontology and Its XML Representation .. 70
  3.2.3 Revising Our Ontology .......................................... 71
3.3 Designing a Truly Useful Ontology ............................... 72
  3.3.1 Beyond the Default Ontology (Flexibility and Extensibility) . 72
3.4 Summary .................................................................. 73

4 Interacting With a DP ..................................................... 75

4.1 Computer Interaction .................................................. 75
  4.1.1 Networking .......................................................... 75
  4.1.2 Security and Privacy ............................................. 78
4.2 Human Interaction ...................................................... 80
  4.2.1 The DP Browser .................................................. 80
4.3 Summary .................................................................. 84

5 Conclusion .................................................................. 87

5.1 Conclusion ............................................................... 87
5.2 Future Work ............................................................ 87

A XML Ontology of Our DP .............................................. 91
List of Figures

1-1 Progressive revelation ........................................... 25

2-1 Relative abstractness of kinds, interfaces, and implementations .... 35
2-2 Containment structure of a DP1Impl ................................ 45
2-3 Class hierarchy for implementations .......................... 47

3-1 Hierarchy for kind CreditCard ............................... 62
3-2 Hierarchy for kind PhoneNumber ............................... 64
3-3 Multiple inheritance of interfaces ........................... 65
3-4 Complicated kind subtree ........................................... 66
3-5 RealLifeObject and DataObject trees ......................... 67

4-1 Network architecture for using a DP via RMI .................. 76
4-2 Viewing information using the DP browser .................... 82
4-3 Editing information using the DP browser ..................... 83
Chapter 1

Introduction

1.1 The Problem With Personal Information Applications

These days, everyone has a lot of personal information. From addresses and appointments to passwords and frequent flyer numbers, it is becoming increasingly difficult for people and computers to keep track of everything [3]. One reason for this is that a person usually stores information in many different places. Some information might be in a Personal Information Manager (PIM), some might be in a cellular phone, and some might be in an email application. The more places that information exists, the harder it is to find the information and maintain any duplicate copies.

The reason why information exists in all of these places is simple: software applications need information to complete their tasks. Each application which uses personal information has its own specialized view of the world. This view is determined by the personal information application’s (PIA’s) intended use. For example, the information which is stored about a person in an address book depends on where the address book is located. If the book is in an email application, then a person is no more than an email address. If the book is in a cellular phone, then a person is no more than a phone number. These two address book entries have no sense of whether they are describing the same physical person or not. This lack of knowledge underlies
why duplicate information is so difficult to maintain. If one person were described by entries in two different address books and the entries recognized that they described the same person, then they would know to synchronize the information in the entries.

In short, the problem with PIAs is that they artificially fragment (and sometimes duplicate) information. In reality, each person is unique and has his own set of personal information. Unfortunately, PIAs do not see a person as a singular being with a set of information. They see only the parts of a person which interest them, not the entire person. This causes many problems with respect to finding different pieces of information, correlating them, and synchronizing them.

1.2 The Solution

1.2.1 Overview

The solution is to specify a place, one per person, where all of that person's information can be accessed. This information gateway would provide a uniform access mechanism whereby each PIA could receive the information it needed. In a sense, the information given out by the gateway would be "filtered" in a different way for each PIA. This filtering would allow the gateway to create multiple views of the information it contained. By using the gateway, all of a person's information could be held together without forcing each PIA to store all of a person's information and abandon its specialized view of the world. Information accessed through the gateway would be, by definition, correct and current.

Imagine some of the benefits which would be provided by a gateway such as this:

- Information for a particular person would be easy to find, since it would all be located in just one place.

- Each PIA could interact with the gateway to get only the information it needed.

- PIAs could synchronize information with the gateway instead of with each other.

This would drastically reduce the number of synchronizations needed. For ex-
ample, ten pairwise synchronizations (PIA to PIA) are needed for a group of five PIAs. If, instead, each PIA synchronizes with the gateway, only five synchronizations are needed. In general, for $n$ PIAs, $\frac{n(n-1)}{2}$ pairwise synchronizations are needed, while only $n$ gateway synchronizations are needed.

- A person who wants to change his information would need to change it in only one place.

- Each person would be associated with an aggregate set of information.

It is important to note that the mere existence of such a gateway does not automatically solve the problems of aggregate information, removal of duplicate information, and synchronization. Rather, these problems can be solved by applications which use the gateway via the gateway’s API.

**Past Work**

A person’s need to replicate information in many places while keeping copies synchronized is discussed in [14]. The requirements of a possible software/hardware solution to these (and other) personal information management problems are discussed in [13]. The system which is theorized in [13] is similar to a DP in that it is designed to give access to all of one’s personal information. The two systems are also designed to communicate with other electronic devices and store information in a flexible way. The authors of [13], however, suggest that the system physically store all of the person’s information and be carried around by the owner, similar to an “all-in-one” personal digital assistant (PDA). The DP, on the other hand, is designed to stay in one place and act as a gateway to information which can be stored elsewhere. PIAs would access the DP’s information from afar.

**Solving the Information Transfer Problem**

Before we delve into the issues involved in designing such a gateway, let us examine how its existence would help solve the information transfer problem.
As mentioned previously, PIAs frequently contain the same types of information. For example, both PIMs and cellular phones store pairs of names and phone numbers. Each PIM and each phone should contain a full phone list because that information to accomplish their tasks. Phones were designed to store phone numbers so that their owners would not have to look up the numbers elsewhere whenever they wanted to make calls. Likewise, PIAs were designed to store most of someone’s information so that, when owners needed a phone number, they would not have to take out their phones to find it.

This example involving phone numbers, PIAs, and phones illustrates the following statement: For data to be truly useful to its owner, it should be useful across devices and applications. After all, each person has his own set of information. Each PIA may show only a small part of that set, but the parts that are shown should be consistent across devices. A person’s choice of which device to use should involve only the characteristics of the device, not whether it has all of the person’s information or not. It should be assumed that all of a person’s devices contain what they consider to be a complete set of information.

In order for this to happen, name and number pairs which are entered into a person’s PIM should eventually appear on his phone and vice versa. Unfortunately, since PIMs and phones have different internal representations of data, this information transfer is very difficult. To transfer information between two PIAs, a specialized connection must be designed which takes into account the two data representations and knows how to map the fields of one PIA into the fields of another PIA. Data which is not transferable between PIAs can be thought of as “trapped” within a single device, application, or data model. Trapped data is not truly useful to its owner.

The number of connections which would be needed to transfer information between all possible pairs of PIAs is mind-boggling when we consider how many PIAs there are, from PIMs and email applications to phones and pagers. There must be a better solution.
1.2. THE SOLUTION

Scenario: Transferring Information Between a PIM and a Phone

Now, let's see how our gateway can help people transfer information between a PIM and a phone. Imagine that you currently own a PIM which stores all of your phone numbers. You now buy a cellular phone and want to copy the phone numbers from the PIM to the phone. Chances are, there are no products available which can link these two devices. You are left with two choices: you can enter all of the numbers manually into the phone, or you can find the numbers in the PIM whenever you need to make a call. Neither solution is very appealing.

Now assume that you have your own personal information gateway. Instead of having to worry about linking the PIM and the phone, you can just have the phone contact the gateway and get the information from there. Since the gateway will have the most current information, the phone will be ready to go once it has copied the information from the gateway.

This scenario applies to any situation in which a PIA needs to obtain information. Perhaps you have entered information into a PIA which you want copied to other PIAs. Or perhaps you just bought a new PIA, which contains no information at all. To update the PIAs, simply have them contact the gateway.

1.2.2 Designing the Solution: Features of the System

We will now look more closely at the requirements and features of our information gateway. The first requirement of the system is that it somehow be able to "filter" information into different formats. This is necessary so that the system can interact with different PIAs. Since the number of different PIAs is likely to keep increasing, the system should not rely on the details of each PIA's internal data structures. Instead, the system should be sufficiently general so that it can handle different PIAs easily. The goal of the system is to be flexible and forward-looking.
Overview

This system, which we call a Digital Persona (DP), acts as a single mechanism for interacting with a set of personal information. A DP, unlike a PIA, does not necessarily store the information in itself. Instead, a DP offers access to information which can be stored in a variety of ways. Rather than being a centralized repository of information, the DP is a more like a central place to interact with information. The DP’s API protocol unifies the semantics which are used to interact with the information. PIAs can interact with the DP using the DP API via a headless interface.

Since a DP is the central place to interact with a person’s information, the information it returns is, by definition, the most correct and currently available. PIAs can synchronize information with a DP rather than with other PIAs, thereby solving the information transfer problem.

Representing Information as Objects

The guarantee that a DP will return correct information may seem a bit unsettling at first. It would appear that this promise could be kept only by synchronizing the information stored in a DP very often with all of a person’s PIAs. This mass synchronization is not necessary, however, since a DP does not have to store any information at all. One of the key features of a DP is that it represents information as objects, rather than as text strings. These objects can contain either information or references to information.

Consider, for example, an object which stands for a person’s postal address. At the most basic level, the object could have only one field, which holds a string representation of the address, and only one method, getAddress, which returns that string. At this point, the DP mostly collapses into a system which stores information as text, with no added functionality. In this way, the DP’s “base case” implementation is simply a text-based system. A DP like this would indeed have to be synchronized with PIAs often if it is to contain current information. The synchronization would update all of the strings inside the objects.
Since the DP represents information as objects, however, an address can have methods associated with it. For example, a more sophisticated implementation of getAddress might have the DP contact a Lotus Organizer application (on the same machine or perhaps over a network) to obtain the necessary address. This would be the implementation to use if the DP's owner knew that the Organizer application always had the most current information.\(^1\) Some sort of transaction management will also have to be used here, in case the DP tries to get information from a source which is currently changing. Implementation details such as this will differ based on the system requirements, such as whether cached values are okay to return or not.

A DP can be guaranteed to always return current information by referencing information which is stored elsewhere and using remote method invocation (RMI) to interact with it. For each object that represents a piece of information, the implementations of its methods can be specialized based on where the current version of that information is stored. The current version may be stored in the DP itself, it may be stored in another application on the same device, or it may be stored on a completely different device.\(^2\) Notice that each object is independent of the others in terms of where it thinks its current information is located. Just because a DP stores phone numbers in the DP itself does not mean that it has to store addresses in itself, also. In this way, the information given out by a DP can be truly decentralized. The same API is used to interact with a DP no matter where the DP gets the data it returns.

Another benefit to representing information as objects (and thereby being able to associate methods with them) is that methods can perform useful computations. For example, a DP might have an object which represents a date. The date object

---

\(^1\)When a method implementation calls for information to be obtained from another device or application, it is frequently necessary to cache data. Caching data helps decrease communication latency (when the target device is busy or far away) and always guarantees a response (even if the target device is turned off). Specific caching strategies and their suitability for different situations will not be discussed in this document.

\(^2\)If the DP needs to contact a device to which it is not physically connected, the DP could either use wireless communications, return a cached value, or not return any value at all. The exact behavior of the DP would depend on the owner's preferences and would be part of the information object's implementation.
could have a method called timeFromToday which calculates the difference between the date stored in the object and the current date, as reported by the device which is storing the date object. Another example involves string manipulation. Assume that an object has one field, which contains a string. The object could have methods which were able to translate the string to different languages. These methods might be called toEnglish, toFrench, and toGerman.

Reflection

Another key technology which is used in the DP system is reflection. In general, a language which supports reflection can answer questions about itself and possibly change its own behavior [16]. Many types of languages have been developed with reflective architectures, including procedure-based, logic-based, and rule-based languages. The first object-oriented language to incorporate an explicit and uniform architecture for reflection was 3-KRS [16].

We will use Java's reflective capabilities to explain reflection. All reflective languages have internal structures which represent the language's domain (i.e., what types of data can be represented by using the language and how they can be manipulated). In Java, these structures are objects which represent classes, methods, fields, and constructors. These objects are called metaobjects, to distinguish them from "normal" objects. A normal object contains information about an entity (such as a person) which is represented by the object. Metaobjects, on the other hand, contain information about normal objects, such as how they are implemented and how the system should handle them.

Each metaobject, just like a normal object, is associated with a class. Some of Java's metaclasses (metaobject classes) include Class, Method, Field, and Constructor. Class is part of Java's java.lang package, while the other metaclasses are part of the java.lang.reflect package. Together, Java's metaclasses form the Java Reflection API.

In truly reflective systems, metaobjects can be modified. By modifying metaobjects, users can incrementally change a language's behavior and implementation to
better suit their needs. These decisions about how a language should behave are encoded in a protocol (a behavioral specification) which operates on the metaobjects. This protocol is called a metaobject protocol (MOP). For more information on MOPs and why they are useful, see [8] and [7].

Java is not truly reflective because its metaobjects and metaclasses cannot be changed or extended. Thus, the behavior of the Java language cannot be changed by using only the Reflection API.³ Java’s reflection mechanism is only meant to provide typing information to the programmer. (This typing information is somewhat similar to that which is provided by C++’s Runtime Type Information (RTTI).) For example, reflection allows a Java program to query an object or a class at runtime to learn about its fields, methods, and any interfaces which it supports. Since Java reflection merely gives a program additional type information, it is sometimes referred to as introspection. For the remainder of this document, the term “reflection” will refer to Java’s version of reflection (namely, introspection).

Interfaces

Interfaces also play a big part in the DP’s information object model. An interface is a collection of method signatures. A class which implements an interface supplies implementations for the methods in the interface. In this way, interfaces separate method implementations from method signatures and behavioral guarantees. [9] discusses how to design interfaces such as these.

An object which is associated with an interface is guaranteed to exhibit certain behaviors when methods in that interface are called on it. The usefulness of guaranteeing method behavior, apart from actual implementation, is discussed in [12]. In particular, [12] mentions guaranteeing method behavior which is defined on an interface instead of just a class. The DP system uses this relationship between interfaces and method behavior guarantees.

In the DP system, interface methods are frequently used to get access to objects

³It is possible, however, to build a MOP on top of Java. This would allow the behavior of the Java language to be changed.
which are contained in other objects. For example, an object representing a person might be specified as returning a String representation of the person’s name if the method getName were called on it. It would not matter what the name object’s class was, as long as getName produced the promised value. Objects which are implemented on different platforms can be dealt with in a similar fashion by using methods defined in their interfaces.

Interfaces allow the DP to ignore where objects come from and how they are implemented and instead focus on what’s important: the information which they contain. For example, a phone number object on a cell phone and a phone number object on a PIM could both support the same interface methods. Interfaces provide flexibility in objects’ inner structures and implementations, including fields and methods.

1.3 The Information Object Model

The information object model involves how information is represented using objects, including use of inheritance and other object-oriented concepts. Information can be represented in many different ways by using objects. One very important aspect of any object model is its balance between structure and flexibility, both of which are useful in object-oriented systems.

1.3.1 Structure and Flexibility

For information to be useful, it must be stored in a well-defined and organized way. Otherwise, it will be very difficult to find or to change a specific piece of information. For example, consider what would happen if you tried to look up a word in a dictionary whose entries were not alphabetized. Searching through the entire book to find the one entry would be ridiculous, a complete waste of time. Dictionaries clearly need a way to organize entries.

Dictionaries also need structure within each entry. Even though entries differ, they each have some parts in common: pronunciation, part of speech, and definition. If the pronunciation marks differed between entries, the dictionary would be more
difficult to use.

The situation is similar for software systems which store objects. First of all, structure is needed in terms of how objects are related to each other (e.g., object A contains object B, object A and object B point to object C, etc.) Secondly, the objects themselves also need structure. This structure comes from the classes that were instantiated to form the objects.

There are many benefits to organizing and forming objects in a structured way. Unfortunately, many systems mandate structure at the expense of flexibility. Flexibility is very important when building a system which needs to handle many different data types, some of which have not been invented yet.

Let us consider the issue of flexibility in the realm of paper address books. Entries in old address books do not have slots reserved for email addresses, cell phone numbers, and fax numbers. A person who wants to store this type of information an old address book would have to come up with his own system for doing so. For example, he might put the information in an unused slot or give each person two entries. A more flexible address book entry design might have some extra unnamed slots, which would make it easier to accommodate new types of information.

Too much flexibility can also cause problems. For example, a really flexible address book entry design might have no slots at all, just space to write in a person’s information. While this design allows each entry to store many different phone numbers, it does not indicate which numbers are for a landline phone, a cell phone, or a fax machine. Both flexibility and structure are needed to make information useful.

Flexibility in a software system is similar. Abstracting away the actual implementation of a method or object from its meaning helps a system treat information from different devices in a similar way. Flexibility in organizing objects comes from an application not relying on a strict, pre-programmed object memory map.

The DP system needs to handle information which is stored in many different data structures. For the system to be useful, it must have the right balance between structure and flexibility. Without structure, the information will be difficult to locate and use. Without flexibility, the DP will be trapped in time, able to interact with
only certain information representations.

1.3.2 Progressive Revelation: The Best of Both Worlds

Progressive revelation (PR) gives the DP system the right balance between structure and flexibility. As you will see, reflection and interfaces are used frequently during PR. Before we can delve into PR, we need to cover kinds and containment.

Kinds

The DP system uses objects to represent information. Each object belongs to one category, which we will call a kind. In the realm of personal information, kinds might include people, addresses, and phone numbers. Kinds abstract away the characteristics of an object, including which methods you can call on it, the values that are stored in its fields, and even the fields themselves. Objects which are of the same kind represent similar types of information (such as addresses) or similar real-life objects (such as people). It does not make sense to compare objects which are of different kinds. For example, an address object cannot be equal to (or unequal to) a person object. They are too different to compare. Two person objects, on the other hand, can be compared.

Associating each object with a specific kind helps a programmer show his intent when creating new objects. For example, if a certain method takes an address as a parameter, it would probably make no sense for that same method to be able to take a person instead. In this way, kinds provide a sort of rudimentary type-checking.

Containment

When we say that one object contains another object, we mean that the first object has some sort of a link to the second object. In our object model, containment can happen in one of two ways: compound objects or "pointers" to objects.

Compound objects are objects whose fields contain values that are also objects. For example, an object which has a field containing a String is a compound object.
This is the simplest type of containment. "Pointers" to objects are not pointers in the classical sense. Rather, they are more like references. They represent a way to access an object which is stored elsewhere, perhaps in a PIA. The example of obtaining an address object by contacting Lotus Organizer is a pointer to an object. In this manner, a single object might be "contained" (referenced) by multiple objects at the same time.

When giving out information, the DP does not distinguish between whether the information came from within a compound object or from somewhere else. As far as the information recipient is concerned, one object was just contained in another object, whether it was due to the container object being a compound object or having a pointer to another object.

The persistence of contained objects can be understood based on the fact that Java, our implementation language, is garbage-collected. If an outer object gets deleted, then it will no longer have a reference to the object which it contained. If the outer object had the only reference to the inner object, then deleting the outer object would also delete the inner object. This would usually occur if the inner object was contained via a compound object relationship. In the case of a pointer to another object, the inner object would not be garbage collected, since it never actually resided in the DP itself. The outer object and its pointer would be deleted, but the inner object would remain intact in its original location.

When an object refers to its inner object, it refers to it by its kind, as opposed to the class from which it was instantiated. This generality allows for more flexibility. For example, a Person object might contain an object of kind Address, representing that person's address. The Person object would not rely on the fact that the address was of a particular class, only that it was of the kind Address. Using kinds to specify containment makes the inner object "pluggable": different types of Address objects can be used within the Person object, as long as they are of the kind Address. This idea is explored further in Section 2.2.2.
Progressive Revelation Itself

Progressive revelation uses kinds, containment, interfaces, and reflection to provide a flexible, extensible way of organizing objects. PR is akin to lazy evaluation in that object structures are discovered at runtime as it becomes necessary. Even though a program starts running without knowing an object’s structure a priori, it can still learn about the object at runtime and handle it effectively by using PR.

Here is an example of progressive revelation, step-by-step. We begin with an object, A, which we know nothing about. The goal is to discover what types of objects A contains. We then repeat the process on those objects, if there are any. While reading this section, it may help to follow along with Figure 1-1.

- Find out which interface A supports by using reflection.
- Note the return values of the interface methods which give access to contained objects. These return values are the kinds or types of the objects which are contained in A. (The return values will be types only if the contained object is a basic Java object such as a String or an Integer.) At this point, we have determined A’s structure one level deep.
- Call the interface methods on A to obtain the objects which A contains. Repeat these three steps (with each contained object in place of A), until all contained objects are basic Java objects. After the process has bottomed out, A’s entire structure has been determined.

Together, interfaces, kinds, and PR allow object containment structures to remain somewhat unspecified until runtime. This gives the DP system its flexibility in organizing objects.

1.3.3 DP Programming Style

Progressive revelation and the DP object model suggest a certain programming style that should be used when interacting with a DP. The idea behind the style is to state all of your assumptions clearly when calling methods on an object. If your
Figure 1-1: Progressive revelation
assumptions are false or the method call otherwise fails, you should then use that information to approach the object in a different way.

The assumptions mentioned above have to do with interfaces and methods. Recall that methods in an interface have specific behaviors associated with them. These behaviors are guaranteed as long as the object supports the interface. Programs which want an object to exhibit a certain behavior should pass to the object not only the method name but also the interface with which it is associated. Stating assumptions in this way can avoid the confusion created by methods which have the same name. If the interface were not passed, then the programmer would have no idea whether the object executed the intended method or just another method of the same name which the object happened to support.

A sample method call might look like this:

\[ \text{APerson}.\text{getObject}('\{\text{Person1}\text{name}\}') \].

In this case, APerson is an object of the kind person. The word in the braces is the name of an interface. In this example, the program wants to get the “name” object which is inside of the person object. The way to do this is to call the method getName. The behavioral specification of this method is attached to the Person1 interface. In other words, getName is guaranteed to return the person’s name only if APerson supports the Person1 interface.

When the method is called, APerson will be tested as to whether it supports the Person1 interface. If it does not, then the program has assumed falsely. A checked exception will be thrown to tell the program which interface is not supported.

The program does not have to specify an interface. If it does not, then the getName method will be called on APerson, regardless of whether this method is part of the Person1 interface or not. If APerson does not support a method named getName,

---

4 The method call \text{getObject(}"\text{name}\text{"}) translates to the method call \text{getName}. This translation will be covered in Section 2.3.

5 In Java, a checked exception must either be declared in the signature of the enclosing method or be caught and handled by using try and catch statements.

6 The functionality which is triggered by these interface statements is part of the DP system implementation. It is not part of Java.
then a checked exception will be thrown to tell the program which method is not supported.

For clarity, the majority of the method call examples used in this document will not include declarations of interface assumptions. However, these assumptions should be stated if a program is to be as robust as possible.

**Handling an Exception**

If a method call throws an exception, then the program must work a little harder to get its task accomplished. Each exception will give some information on why the method call failed.

The two possible types of exceptions are not supporting an interface and not supporting a method. In each case, the name of the interface or method will be included in the exception. Based on information provided by the exception, the program might handle the exception by doing one of three things:

- The program first tries to convert the object to a type that does support that interface and method. (Object conversion is part of the DP system.)

- The program tries to find a similar interface or method that the object does support. The object’s interfaces and methods can be examined by using the Java Reflection API or DP reflection. DP reflection is simpler and is designed to help the program complete this task.

- The program gives up and converts the object into a String object. (String conversion is also part of the DP system.) This should be the program’s last resort.

The more contingency plans a programmer puts into an application, the more robust the application will be. Preparing for failed method calls allows a program to fully take advantage of PR’s flexibility.
1.4 External Management of Information

This section introduces how information in a DP might be used by others. Security issues are also discussed.

Using a DP to Distribute Information

So far, we have tacitly assumed that a person's DP would be accessed only by PIAs which belong to him. But a DP can also be used as an easy way for someone to give information to others. For example, instead of repeating your contact information to many different people, you could instead give them the location of your DP. This way, friends and companies who wanted your information could access it without you needing to do anything. The details on how people and computers can access a DP's information will be covered in Chapter 4.

DP Access Control

Once we decide to let more than one person interact with a DP, we must face the issue of access control: i.e., who will have which types of privileges (read/write) with which types of information.

In general, the DP would have different levels of access restrictions based on who is trying the access which information. For example, users who are identified as friends might have read-only access to a person's address, phone number, and email address. On the other hand, users who are identified as companies might have read-only access to just the person's email address. The owner of the DP would have read and write access to all of the information which is stored in a DP, of course. More specific security options will be discussed in Section 4.1.

1.5 Thesis Overview

The remainder of this thesis is organized as follows: Chapter 2 describes how a DP stores information as objects. Chapter 3 discusses the need for a default object struc-
ture and how this structure can be represented. Chapter 4 outlines how computers and people can interact with a DP. Chapter 5 concludes the thesis and suggests areas of future research.
Chapter 2

DP Internals

As mentioned previously, a Digital Persona stores information in the form of objects. This chapter will discuss the characteristics of these objects and how they are stored within a DP.

2.1 Object Type System

2.1.1 Overview

Class/Instance v. Protoyped

One of the first steps in modeling a system using objects is to decide whether the underlying object system should be class/instance or prototyped. In class/instance systems, each class describes a particular set of fields and methods. An object is formed by instantiating a class. Many objects can derive from the same class. The programmer can tell whether objects are similar by looking at their classes in relation to the overall class structure. Class/instance programming languages include C++, Smalltalk, and Java.

In prototyped systems, each object has its own special type based on the object’s fields and methods. For example, a family object type could have two, three, or four fields depending on how many people were in that family. Giving an object a tailor-made type ensures that no information is lost and no fields stand empty. The
problem here is that similarities between family objects are not recognized. After all, family objects should be somewhat similar, even if they contain a different number of fields. The programmer, on the other hand, doesn’t know how the various family types differ. Giving each family object a different type takes away the programmer’s ability to interact with all family objects in a similar way. Prototyped programming languages include Self [21], Omega [2], and NewtonScript [20].

Which system is better depends on the situation. If many similar objects will exist, then class-instance is probably better. If there will be many different types of objects and not many objects of the same type, then prototyped might be better. The DP system is class-instance because there will be many instances of objects within the same category (e.g., many people objects and many address objects).

**Whether to Have Multiple Classes Per Category**

Objects can be divided into groups, or categories, based on their similarities. For simplicity, these categories should be distinct enough to ensure that each object can belong to only one category. In the realm of personal information, some suitable categories\(^1\) are person, name, address, and phone number.

The next issue is whether objects within the same category should contain the same fields and methods. In other words, should there be only one available class per category? Remember that once objects have been created from a class, the definition of the class (fields and methods) cannot be changed. For example, one cannot instantiate a person class which has a “name” field and then modify the class by taking away the “name” field.\(^2\) If this occurred, then the older objects in the system would not function the same way as the newer objects.

Consider the example of choosing appropriate fields to use for phone-number objects. Suppose a class is defined with one area code field and one number field. Using this class would ensure that no phone-number objects would contain empty fields, since all phone numbers have an area code and a number. However, using this class

---
\(^1\)Eventually, we will use the technical term “kind” instead of the colloquial term “category.”
\(^2\)The values which are stored in the fields can change after the object has been created, of course.
for all phone numbers might result in lost information, such as the international code or extension.

An alternative might be to add two more fields to the phone-number class to handle this additional information. Having a policy of always adding fields to classes to accommodate special cases will ensure that no information is lost. Unfortunately, it will also decrease the system's level of usability. First, it will be more difficult for the user to find the relevant information. Handling special cases may have resulted in redundant fields which serve the same purpose but have different names. Also, some field names may mean different things to different people, which will result in inconsistent ways of assigning values to fields. Secondly, there will probably be many objects with empty fields. Not only will these empty fields waste memory space, but they will also make programs more cluttered, since each field will have to be tested as to whether it is null.

Building some flexibility into the class system will avoid these problems. For example, we can use inheritance and class hierarchies to create new classes that are related to existing classes. Our solution to this problem will be covered in Section 2.1.2.

2.1.2 Kinds, Interfaces, and Implementations

Choosing a Language

The object type system we designed relies on two key concepts: interfaces and reflection. Interfaces are collections of only method signatures (no implementations) which can be associated with a particular class or type. Classes which supply these implementations are said to implement the interface. Classes can implement more than one interface at a time. Reflection is the ability to look inside an object and inspect its fields and methods at runtime. For more information on reflection and interfaces, see Section 1.2.2.

Our design could have been implemented in many different languages. We chose to use Java because it already had the capabilities needed for interfaces and reflection. C++, on the other hand, did not have the same kinds of interfaces or reflection. We
could have simulated reflection in C++ by adding it to the language, as done in ObjectStore [5], but we felt that Java was more suitable. Also, we liked the fact that Java is platform independent, since the DP code has to run on many different devices to be truly useful.

Overview of the System

The kind/interface/implementation system is a way of thinking about the objects which are stored in a DP. The system is designed to be extensible while allowing the programmer to manipulate different objects in meaningful ways. Inherent in the system are ways to classify and create objects.

Each object in a DP is associated with exactly one implementation, at least one interface, and only one kind. Kinds are the most abstract part of the DP ontology, while implementations are the most concrete. Together, an object, its interface(s), and its kind form a special hierarchy. See Figure 2-1 for more details.

Kinds

We use the term “kind” to represent more precisely an object’s general type or category. Some kinds in the DP ontology are Person, PostalAddress, and Date. The name of an object’s kind helps the programmer understand what the object represents. Two objects that have the same kind are not guaranteed to be similar in terms of their fields or methods. A kind is simply a label that indicates intent. Kinds divide the namespace for interfaces.

In Java, we represent kinds as marker interfaces. That is, kinds are interfaces which define no methods. (Interfaces are Java data types that are similar to abstract classes.) The Java code describing the kind Person would look like this:

```java
public interface Person { }
```

---

3For the remainder of this document, the italicized word “interface” refers to the Java data type, while the non-italicized word “interface” refers to the DP type system.
Figure 2-1: Relative abstractness of kinds, interfaces, and implementations
A class which implements a marker interface needs only to declare the interface in its implements clause. No new methods need to be added to the class. The fact that a class implements a particular marker interface provides additional information about an object of the class or the class itself. Reflection can be used to determine which interfaces a particular class implements.

Interfaces

An interface is a set of methods. Each interface is associated with exactly one kind. The methods in an interface are meant to be applied to objects of the parent kind. For example, methods in the interfaces Person1, PostalAddress1, and Date1, would be applied to objects of the kinds Person, PostalAddress, and Date. (Conventionally, interface names begin with the kind with which they are associated.)

Assume that an object is associated with a particular interface. This means that all the methods in the interface are well-defined for that object. In essence, associating an interface with an object tells the programmer what he can "do" with the object. The programmer is guaranteed a certain level of functionality when calling one of these methods on that object. Interfaces are somewhat useless unless the programmer knows the functionality associated with each method. Sometimes the functionality is obvious. At other times, interfaces must be accompanied by documentation that describes their methods.

In terms of Java implementation, interfaces are similar to kinds. Interfaces are implemented as interfaces which extend their parent kind and contain method signatures. (Interfaces representing kinds do not have any method signatures.) Interfaces are divided into smaller namespaces according to which kind they extend (i.e., their parent kind). Let's create an interface called CreditCard1 which extends the kind CreditCard. CreditCard1 has methods which allow the programmer to get and set the account number. The Java code for CreditCard and CreditCard1 would look like this:

```java
public interface CreditCard { }
```
2.1. OBJECT TYPE SYSTEM

public interface CreditCard1 extends CreditCard {
    public String getAccountNumber();
    public void setAccountNumber(String acctnumber);
}

Interfaces can also inherit from one or more interfaces. This is one of the key ideas behind the DP's extensible object type system. For example, let's return to the phone number problem that was mentioned in Section 1.2. Assume that in the past, a programmer defined a basic interface for a phone number object, PhoneNumberBasic, which contains only the methods getAreaCode, setAreaCode, getNumber, and setNumber. Later on, he wants to be able to handle international codes and extensions. He can't redefine the PhoneNumberBasic interface, since it is being used by other objects. Instead, he should define another interface, PhoneNumberAdvanced, which would include four new methods (getIntCode, setIntCode, getExt, and setExt) and inherit another four methods from PhoneNumberBasic. In this way, the programmer is guaranteed the ability to call at least eight different methods on any object which implements PhoneNumberAdvanced. The Java code for PhoneNumberBasic and PhoneNumberAdvanced would look like this:

    public interface PhoneNumberBasic {
        public String getAreaCode();
        public String getNumber();
        public void setAreaCode(String x);
        public void setNumber(String x);
    }

    public interface PhoneNumberAdvanced extends PhoneNumberBasic {
        public String getIntCode();
        public String getExt();
        public void setIntCode(String x);
        public void setExt(String x);
    }

Of course, interfaces do not have to inherit from other interfaces. If the programmer wants to create an interface that is different from all existing interfaces of that
kind, then he can have the interface descend directly from the parent kind, just as CreditCard1 descends directly from CreditCard.

Whenever possible, programmers should try to have new interfaces inherit from existing interfaces. This way, code which handles objects that implement the older (super) interface can also handle objects that implement the newer (sub) interface. Of course, the new interface should inherit from the existing interface only if this object substitution would make sense. When the interfaces are radically different, one interface should not inherit from the other.

In the DP, interfaces are frequently used to give programmers access to values stored in an object’s internal fields. This topic will be discussed further in Section 2.2.

Implementations

Implementations are the most concrete part of the kind/interface/implementation hierarchy. Essentially, an implementation is just an object’s data type, the class that was instantiated to form the object. The concrete quality of an implementation sets it apart from its associated kind and interface(s). While kinds and interfaces are implemented as Java interfaces, implementations are actual Java classes that can be instantiated.

The implementation of an object tells the object how to perform the methods that are defined in the object’s interface(s). If an object is associated with an interface, then the object’s implementation must have method bodies for each of the methods mentioned in the interface. Since there are many different ways to write code that provides a certain method behavior, there can be many different implementations associated with the same interface.

All implementation classes, including those representing people, addresses, and even the DP itself, “contain” other objects. (See Section 1.3.2 for more details.) Some implementation classes are Person1Impl, PostalAddress1Impl, and Date1Impl. (Conventionally, implementation names consist of an interface with which they are associated, suffixed with “Impl.”)

The implementations we created for our DP were very simple. They included
methods which could get or set values in an object's fields. These values were stored in the DP itself. But implementations can also have methods which return values that are computed or retrieved from elsewhere. For example, a person implementation could contain a method named getAge which calculated a person's age based on the person's birthdate (stored in the object) and the current date (obtained by constructing a Java Date object). A getAddress method could obtain an address by contacting another computer over the network and querying one of its applications (e.g., Lotus Notes or Microsoft Organizer).

In Java, an implementation is a class. The class lists all of its fields and provides the methods mentioned in its interface(s). Here is the Java code for the Date1 interface and the associated Date1Impl implementation:

```java
public interface Date1 extends Date {
    public Integer getYear();
    public Integer getMonth();
    public Integer getDay();
    public Integer getHour();
    public Integer getMinute();
    public Integer getSecond();
    public Integer getFractionSecond();
    public String getTimeZone();
    public void setYear(Integer x);
    public void setMonth(Integer x);
    public void setDay(Integer x);
    public void setHour(Integer x);
    public void setMinute(Integer x);
    public void setSecond(Integer x);
    public void setFractionSecond(Integer x);
    public void setTimeZone(String x);
}

public class Date1Impl implements Date1 {
    Integer year = null;
    Integer month = null;
    Integer day = null;
```
CHAPTER 2. DP INTERNALS

```java
Integer hour = null;
Integer minute = null;
Integer second = null;
Integer fractionSecond = null;
String timeZone = null;

public Integer getYear()
    { return year; }
public Integer getMonth()
    { return month; }
public Integer getDay()
    { return day; }
public Integer getHour()
    { return hour; }
public Integer getMinute()
    { return minute; }
public Integer getSecond()
    { return second; }
public Integer getFractionSecond()
    { return fractionSecond; }
public String getTimeZone()
    { return timeZone; }

public void setYear(Integer x)
    { year = x; }
public void setMonth(Integer x)
    { month = x; }
public void setDay(Integer x)
    { day = x; }
public void setHour(Integer x)
    { hour = x; }
public void setMinute(Integer x)
    { minute = x; }
public void setSecond(Integer x)
    { second = x; }
public void setFractionSecond(Integer x)
    { fractionSecond = x; }
public void setTimeZone(String x)
    { timeZone = x; }
```

Just as interfaces can inherit from each other, so can implementations. In fact, interface inheritance and implementation inheritance are closely linked. Recall that an interface descending from another interface defines only those methods which are new to it. The descendant interface’s implementation could likewise descend from the parent interface’s implementation. For example, assume we have the interface Person1 and its associated implementation Person1Impl. We could then create a new interface, Child1, which descends from Person1 and adds some of its own special methods. The implementation associated with Child1, Child1Impl, could then descend from Person1Impl. That way, Child1Impl would have to provide method bodies for only those methods which it did not share with Person1Impl (namely, the new methods that were added in the Child1 interface.)

Since implementations are Java classes, they must eventually descend from the Java class Object. In the DP, however, we have added some extra classes in between
the Object class and our Impl classes. These extra layers are the subject of Section 2.3.

Each implementation should be associated with at least one interface. Otherwise, a program will not be able to use the special reflection methods which are in the DP API.\textsuperscript{4} Among other things, these methods allow a program to easily determine what types of objects are contained in a particular object. The use of reflection in the DP system will be discussed further in Section 2.3.

Subclasses and Subtypes

Before we leave the topic of implementations, a few words should be said about the two types of inheritance: inheritance of implementation and inheritance of behavior. In the realm of implementation, inheritance helps a programmer create new classes by reusing the code of existing classes. This is called a subclass relationship. In the realm of behavior, inheritance is used to establish an “is-a” relationship between objects. This is called a subtype relationship.

In the DP system, objects which descend from other objects generally use the parent object’s implementation of the inherited methods. This means that the descendant object is a subclass of the parent. The programmer can also choose to override some of the inherited methods and give them new implementations. Depending on how many inherited methods are overridden, the descendant may or may not be considered a subclass.

When a descendant class is designed to exhibit the same behavior as its parent, then the descendant is called a subtype. This behavioral guarantee means that an object of a subtype can be substituted anywhere for an object of the supertype. This is known as the principle of substitutability [6].

While it is primarily up to the programmer whether or not he uses implementation inheritance, inheritance relationships involving behavioral properties are vital to the DP object model. This is due to the DP’s reliance on interface methods having well-defined behaviors. Since descendant classes inherit the interfaces of their parents,

\textsuperscript{4}Even if an object has no interfaces, a program can still examine it using the Java Reflection API.
they must supply implementations for these methods. The DP system relies on the fact that interface methods have the same basic meaning, no matter which class is implementing them. Because of this, the descendant’s implementations must preserve the method behavior that is guaranteed by the interface. The code can either come from the parent (in which case the descendant would be both a subtype and a subclass) or be completely rewritten, thereby overriding that of the parent (in which case the descendant would be a subtype only). If the new implementations produce drastically different behaviors, then the DP system will be rendered unusable. In the DP system, a descendant class must always be a subtype of its parent.

2.2 Structure Surrounding Objects

Now that we have covered how objects in a DP are specified and created, it is time to address the issue of how they are organized within a DP.

2.2.1 Motivation

As stated previously, a DP will “contain” other objects. These objects will be kept in some sort of data structure. There are two important issues to consider when choosing an appropriate structure: simplicity and “findability” (how easy it is to find objects).

Consider what would happen if all of the objects were stored, disorganized, in a list structure such as an array. It would be very simple to add objects to the system, but very difficult to find one particular object. Keeping track of object indices would probably be difficult, since adding objects to or deleting objects from the array might change the other objects’ indices. This would lead to many problems for the programmer. Using an array to store objects is too brittle of an architecture. The objects should be stored in a more robust, extensible way.
2.2.2 Interfaces and Containment

We wanted to make the internal structure of a DP as extensible as possible. We decided to base our structure on the concept of "containment." As stated in Section 1.3.2, containment can occur in one of two ways: compound objects or "pointers" to objects. Semantically, we think of a contained object as being "inside" an outer object, even if the contained object is located on an entirely different device.

The way the programmer knows what is "inside" an object is by looking at the object's interface methods. In the DP implementation, interface methods are frequently used to give the programmer access to the "inside" of an object. In other words, they allow the programmer to get and set values of contained objects. These contained objects can either be stored in fields of the original object, or they can be located on another device. For example, Person1 interface methods include getName and setName, while PostalAddress1 interface methods include getCity and setCity. These method signatures look like this:

```java
    PersonName getName();
    void setName(PersonName x);
    String getCity();
    void setCity(String x);
```

Get/set interface methods return values/take arguments that are either basic Java objects (String, Integer) or kinds (PersonName, Date, PostalAddress). Java interfaces and classes (representing DP interfaces and implementations) should not appear as return values of or arguments to get/set interface methods. This aspect of interfaces is central to the DP's extensibility.

For example, say that we define an object that stands for an organization. The organization object contains an object representation of the organization's president. The president object can be set by passing a person object to the interface method setPresident. Since the only requirement of the president object is that it represents a person, the signature of setPresident should require an object of kind Person.

In order to follow the DP's design goal of flexibility, the organization object must
not rely on the fact that its president object is a certain type of Person. This generality is programmed into the organization object by requiring the setPresident method to accept any Person as an argument, not just a Person of a specified interface or implementation (such as Person1 or Person1Impl). Unnecessarily constraining the type of the president greatly decreases the internal flexibility of the organization object. If the organization object really needs a certain type of Person to be its president, then the programmer might want to create a new kind. Situations like these are good arguments for the development of a kind hierarchy, with kinds inheriting from other kinds. This hierarchy would be similar to the inheritance hierarchy that is now possible in the DP system. In the current DP implementation, kinds exist separately and are not connected in any way.

This example shows why get/set methods deal with kinds, as opposed to interfaces and implementations. When the interface to a container object specifies a kind for the type of an interior object, there are two main benefits: 1. The container object is as general as possible about the interfaces and implementations of the objects it contains. 2. The programmer can still reason about what those objects are, based on their kind (since the kind is a label that indicates intent).

Progressive revelation (PR), which was mentioned in Section 1.3.2, is directly related to interfaces and containment. Recall that PR is the method of peeling back the layers of an object one-by-one, in order to determine exactly what that object is and what objects it contains. PR would be impossible if there were no way for the programmer to determine which objects are contained in another object. Get/set interface methods fill this need. The mechanics of PR, including how it is performed using the DP API, will be discussed in Section 2.3.

### 2.2.3 The DP Itself: The Outermost Container Object

Now that we know how to construct and use implementation objects, we can build the DP itself. As mentioned previously, the interfaces of implementations determine their containment structure. Since the DP itself also contains objects, "building" the DP is as simple as specifying its interface. The DP’s interface will determine exactly
what types of objects are stored in a DP and how they are organized. Thus, a DP's containment structure can be deduced from examining the interfaces of a DP object and its inner objects.

Figure 2-2 shows the containment structure of a sample DP object. This DP contains two objects, a Person object (its owner) and an AddressBook object (the owner's address book). The AddressBook object contains two Person objects (entries in the address book). Each Person object contains three objects: a PersonName (the name of the person), an Address (where the person lives), and a PhoneNumber (the person's phone number). The Address and PhoneNumber objects also contain other objects. For clarity, not all levels of contained objects are shown in the figure. Each object is labelled with its implementation and its kind.

Figure 2-2: Containment structure of a DP1Impl
2.3 DPObjec\textbf{t}: The Building Block of a DP

In this section, we focus on implementations. As mentioned previously, one of the
goals of the DP design is the ability to define new object types and use them in a
meaningful way. The key to this extensibility is DPObjec\textbf{t}, the Java class from which
all implementation classes descend.

\subsection*{2.3.1 DPObjec\textbf{t} and the Class Hierarchy for Implementations}

As mentioned previously, implementation classes do not descend directly from the
Java class Object. This section describes the two levels of classes which separate the
Object class and an implementation class in the class hierarchy. You may wish to
follow along by looking at the class hierarchy illustration in Figure 2-3.

When designing the DP object system, we knew that the Impl objects stored in
a DP would be very similar. Each object would contain other objects and would
have methods which allowed the programmer to get and set the values in of these
objects. The similarities between Impl objects should be exploited. To make this
task easier, all implementation classes descend from a common class: DPObjec\textbf{t}.
DPObjec\textbf{t} is an abstract class which descends directly from the Java class Object.
Some of the methods in DPObjec\textbf{t} are implemented right in DPObjec\textbf{t}, while others
are implemented in the Impl classes themselves. DPObjec\textbf{t} is meant to be the most
general object that is possible in a DP system. Due to DPObjec\textbf{t}'s generality, a
DPObjec\textbf{t} in itself would be useless. That is why DPObjec\textbf{t} is an abstract class.

Implementation classes do not descend directly from DPObjec\textbf{t}, either. They
descend from either RealLifeObject (RLO) or DataObject (DO). RLO and DO are
abstract classes which descend directly from DPObjec\textbf{t} and do not declare any of their
own methods. A class descending from RLO should represent a somewhat tangible
object or entity such as a person or a computer. Classes descending from DO should
represent intangible information, such as an address, a date, or a phone number.
Here are some examples illustrating from which class an Impl class should descend:
Figure 2-3: Class hierarchy for implementations
A person is an RLO. The name of this person is a DO. A building is an RLO. The address of the building is a DO. At the moment, the DP system does not distinguish between RLOs and DOs. We included RLO and DO in the class hierarchy because we feel that this distinction may become important in later projects. See Chapter 5.2 for more information.

2.3.2 Methods Defined in DPObject

DPObject's methods can be divided into four categories, based on their usage: get/set methods, object introspection methods, type conversion methods, and helper methods.

Get/Set Methods

These methods (along with those defined in an object's associated interface) allow the programmer to query and set the values that are stored in an object's fields. DPObject's query and set methods are merely syntactic sugar. They represent more complicated calls involving the object's interface methods. Using the DPObject methods, instead of the interface methods directly, allows the programmer to use the same method for a wide variety of objects. All of these methods must be used on objects which inherit from the class DPObject. The desugaring methods are explained below. Please note the completion of one method call may require many desugarings. Each desugaring step description includes the exceptions which can be thrown during that step. Querying and setting values will be considered separately.

The DPObject methods we will be covering include getObje ct, getStri ng, and setObje ct. Here are their method signatures:

```java
Object getObject(String fieldname);
String getString(String fieldname);
void setObject(String fieldname, Object value);
```

Here is the grammar for the following desugaring rules:
2.3. DOBJECT: THE BUILDING BLOCK OF A DP

Non-Compound Field Name E.g., name, address, last. When the name appears in parentheses, the first letter is not capitalized. When the name appears in a method name (e.g., getName or setName), the first letter is capitalized. DObject uses reflection to obtain the necessary get or set method object. If the programmer wants to include the interface which contains this method, then he should prefix the field name with the interface name in braces. (e.g., PersonName1.first)

Compound Field Name A field name that has two or more parts, separated by periods. E.g., name.first, address.city, Person1name.PersonName1.first.

Period (.) Separator used in compound field names.

Obj An object descended from DObject.

X A non-compound field name.

Y Any field name (compound or non-compound)

Z An object that is of the proper kind or basic Java type, depending on which “set” method it is being passed to.

Querying Methods (getObject and getString)

_Obj.getObject(X) -> Obj.getX()_ The DObject method getObject is desugared to Obj’s corresponding interface method (get*). This step will throw a checked exception if 1) Obj does not have a method called getX which takes no arguments or 2) X contains a false interface assertion.

Example: APerson.getObject("name") -> APerson.getName()

_Obj.getObject(X.Y) -> Obj.getX().getObject(Y)_ This is a more complicated example of the previous rule. X and Y represent where the target object is located in the containment hierarchy. This step will throw a checked exception if 1) Obj does not have a method called getX which takes no arguments or 2) X contains a false interface assertion.
Example: APerson.getObject("address.city") ->
APerson.getAddress().getObject("city")
Example: APerson.getObject("address.zipCode.firstFiveNumbers") ->
APerson.getAddress().getObject("zipCode.firstFiveNumbers")

**Obj.getString(Y) -> Obj.getObject(Y).toString()** The DObject method getString is desugared to calling DObject's method toString on the object returned bygetObject. (DObject's toString implementation may have been shadowed by another implementation in the object's Impl class.) This step cannot throw any checked exceptions.
Example: APerson.getString("name") -> APerson.getObject("name").toString()
Example: APerson.getString("address.city") ->
APerson.getObject("address.city").toString()

**Setting Methods (setObject)**

**Obj.setObject(X, Z) -> Obj.setX(Z)** The DObject method setObject is desugared to Obj's corresponding interface method (set*). This step will throw a checked exception if 1) Obj does not have a method called setX which takes one argument, 2) X contains a false interface assertion, or 3) Z's kind/type doesn't match the signature of setX.
Example: APerson.setObject("address", AnAddress) ->
APerson.setAddress(AnAddress)

**Obj.setObject(X,Y, Z) -> Obj.getObject(X).setObject(Y, Z)** This is a more complicated example of the previous rule. X and Y represent where the target object is located in the containment hierarchy. This step cannot throw any checked exceptions.
Example: APerson.setObject("address.city", ACity) ->
APerson.getObject("address").setObject("city", ACity)
Object Introspection Methods

DQObject offers many methods which help the programmer learn about (reflect on) an object. Some of these methods provide information about an object’s place in the kind/interfaceimplemementation hierarchy. For example, getImpl, getInterfaces, and getKind each return an object (or array of objects) that represents either the object’s implementation (a class), interface (an interface), or kind (also an interface). Here are their method signatures:

```java
Class getImpl();
Class[] getInterfaces();
Class getKind();
```

There are two other methods in this category: getInterfaces and getImps. get-Interfaces tells the programmer which interfaces are associated with a particular kind. getImps indicates which implementations are associated with a particular kind or interface. Here are their method signatures:

```java
Class[] getInterfaces(Class c);
Class[] getImps(Class c);
```

DQObject also offers methods which help the programmer look inside the object itself. These methods, getGetMethod, getSetMethods, and getDpFields, return an Enumeration of the “get” methods, “set” methods, or field names which the object contains. These methods are very useful when it is necessary to iterate through the objects contained in another object.

Type Conversion Methods

Type conversion methods help programs handle objects that have unfamiliar data types. Since the DP’s flexible architecture allows programmers to easily create new kinds, interfaces, and implementations, applications may frequently run across unfamiliar data types. In these situations, an application could either query an object’s
interface to find an appropriate method or use a type conversion method. The type conversion method would translate the unfamiliar object into a type which the system knows how to handle. An application can discover which type conversion methods are supported by a particular object by calling getConversionMethods. This method, which is defined in DObject, returns an array of Method objects which represent the target object's type conversion methods.

For example, imagine that a method returns an unfamiliar object. The encasing application would then use object introspection methods to determine that the object has the kind Person, the interface PersonUnusual, and the implementation PersonUnusualImpl. The application has been written to rely on methods which are defined in the interface Person1. Thus far, there have been no problems, since all Person objects which have been encountered have been Person1Impls, which support the Person1 interface. The program could find out which methods the object supports by reflecting on its interface. (See Section 2.3.3 for more information.) However, deciding at runtime which of these methods to call can be very difficult. Instead, the program could call getConversionMethods and choose one of the returned methods. By calling this method, the program might be able to extract some useful information from the unfamiliar object. For example, a program might be able to convert the PersonUnusualImpl object into a Person1Impl object.

Currently, DObject has one type conversion method: toString. This method produces a String representation of a DObject and shadows the toString method as defined in java.lang.Object. DObject's toString methods is very basic: it iterates through the object's "get" methods and lists each fieldname and the value stored in that field (as a String). When the value stored in a field is a containing object, the String value of that object will not be very helpful. To make toString more useful, we have given every Impl class a toString method which shadows DObject's toString method.

In this way, different Impl objects can be converted to Strings in different ways. For example, a PersonNameBasicImpl object (with the fields first, middle, and last) might translate to a String containing just the values of these fields, in order, separated
by spaces. A DateBasicImpl object might translate itself to a String using a comma to separate the day and year and colons to separate hours and minutes.

**Helper Methods**

The remainder of DObject’s methods are helper methods. They provide functionality related to saving and loading objects and displaying object information in HTML. We will explore some of this functionality later when we discuss the HTML interface to the DP in Section 4.2.

### 2.3.3 Why DObject Works (Java and Reflection)

Instead of explaining how every method in DObject works, this section will concentrate on how DObject uses reflection and the Java Reflection API. The Reflection API is comprised of the Class class in java.lang and the java.lang.reflect package. Reflection gives a program the ability to “inspect and manipulate itself.” [4] The programmer, in turn, can work with a class’ internal structure, such as its constructors, fields, and methods. Fields can be queried and set, and constructors and methods can be invoked.

To illustrate the power of reflection, we will examine its role in the DObject methods `getGetMethods` and `setObject`. Remember that whenever a Class or Method object is mentioned, the Reflection API is being used.

**getGetMethods**

The goal of `getGetMethods` is to return an Enumeration of Method objects which the programmer can use to examine an object’s inner objects. We refer to these methods as the “get” methods, since all of their names begin with the word “get.” (getName, getCity, etc.) Get methods take no arguments. The methods are also public, since they need to be called by different objects.

The implementation of `getGetMethods` consists of these four steps:

1. Obtain the Class object that stands for the class of the target object. (For
example, the Class object of a person implementation object might correspond to a Person1Impl.)

2. Call getMethods on the Class object. This gives us an array of Method objects representing Class’ public member methods. By the way, not all of the Method objects returned by getMethods are “get” methods.

3. Call getName on each Method object. Check whether the name starts with “get.”

4. Call getParameterTypes on each Method object. Check whether the method takes no arguments.

5. Make an Enumeration of the methods that qualified as “get” methods, based on their names and (lack of) parameter types.

setObject

Recall that setObject desugars to either 1) one “set” call or 2) one getObject call and one setObject call. The setObject call, if there is one, can contain a compound field name as an argument. A compound setObject call would then desugar (in one step) to one “get” calls and one setObject call. Here’s an example of a step by step desugaring of a compound setObject call:

• Obj.setObject(“president.address.city”, ACity)

• -> Obj.getObject(“president”).setObject(“address.city”, ACity)

• -> Obj.getObject(“president”).getObject(“address”).setObject(“city”, ACity)

• -> Obj.getObject(“president”).getObject(“address”).setCity(ACity)

setObject uses the Reflection API to obtain the target object’s “set” interface method (e.g., setFirst). We will concentrate on this aspect of the setObject implementation.
The goal of this part of setObject is to obtain the appropriate "set" Method object. This object will have two characteristics. First, it will have a name which matches setObject's fieldname argument. For example, setObject("name", X) would look for a method called setName. Second, it will take a parameter that is valid based on setObject's value argument. (The value argument is the value that we want to assign to the specified field. Here, we represent the value by X.)

For example, if X is a String, then setName should take a String argument. If X is an implementation (Impl class), then setName should take a kind argument that is X's kind. Remember that the appropriate set method might specify a kind, such as a Person, as a parameter. As long as the setObject value argument is of type Person, then we will find the correct "set" method.

First, we get the target object's Class object and call getMethods, as in the getDpMethods implementation. Next, we enumerate through all of the Method objects, checking to see if they have the appropriate name (by calling getName) and take an appropriate parameter (by calling getParameterTypes). We return the method that passes both of these tests.

2.3.4 How DPObjec't Is Used In Progressive Revelation

As mentioned previously, progressive revelation (PR) involves examining an object to determine what types of objects it contains and then recursively examining those objects. PR makes it possible for a program to dynamically determine what types of objects are present. Once objects have been more closely identified, the program can then change its behavior. In this way, the DP system gives the program more flexibility at runtime.

PR is achieved by using different DPObjec't methods. We will now give a step-by-step example which covers how to perform PR by using the DP API. Keep in mind that, at any point, an object's kind, interface(s), and implementation can be determined at any point by calling the DPObjec't methods getKind, getInterfaces, and getImpl, respectively.

Imagine that a program (P) encounters a DPObjec't (D). P uses the method
getImpl to obtain the Class object which represents D’s implementation class. P then calls getGetMethod, which returns an Enumeration of Method objects which represent D’s “get” methods. The return values of D’s get methods correspond to the types of objects which are contained in D. To obtain the return values of the Method objects, P calls getReturnType on each Method. getReturnType returns a Class object representing the type of the return value as it was declared in the method. These return types would be either kinds or basic Java classes, based on the DP style of writing suitably generic “get” interface methods.

P has now determined what types of objects are contained in D. This was just the first part of performing PR. The previous steps can now be applied recursively to the objects inside D. The recursion will bottom out when a get method returns a basic Java object (e.g., String or Integer). This is the base case, since basic Java objects are not DPOObjects. At that point, the program recognizes the Java object and can manipulate it if necessary.

**What Happens When PR Reaches a Dead End**

The main point behind PR is to allow a program to change its behavior based on the type of an object. For example, consider how a program might obtain a String representation of an object of kind Address. The first thing the program might do is check to see if it recognizes the object’s type. If it does, then it would know which method is appropriate, since the method was hard-coded before runtime. For example, an object of type BasicAddress might have a method called addressTo aftermarket, while an object of type ComplexAddress might have a method called entireAddressTo aftermarket.

If the program does not recognize the object’s type, then it can try to deduce which of the object’s methods it should call. In this case, the program might look at the names of the object’s methods and search for related substrings. In this case, the program might look for “string” or “address”. (This task can be accomplished by using the reflective methods getMethods and getColumnName.)

If the program cannot find any appropriate methods, then the program has encountered an object which it cannot handle conveniently. PR seems to have reached
a dead end. This is when DPObj ect’s type conversion methods come in handy. When a program does not know what to do, it can convert the object from an unfamiliar type into a familiar type.

Right now, the DP system has only one type conversion method: toString. When a program does not have a hard-coded way to handle a specific type and cannot find an appropriate method to call, it can call toString. toString has a default implementation in DPObj ect and might have a more specific implementation in the object’s implementation class. Converting the object to a string allows the program to handle or compute on the object, since the program knows how to handle Strings.

There could also be type conversion methods which know how to convert specific object types into other object types. This knowledge is similar to knowing how to map the fields of one object into the fields of another object. For example, assume that we have two interfaces: SimpleName and ComplexName. SimpleName gives access to a first and last name only, while ComplexName also gives access to a middle name and prefix. A program that does not know how to handle a ComplexName might use a type conversion method to convert the ComplexName into a SimpleName. The type conversion method might create a new SimpleName object and set its fields by using values returned by calling getFirstName and getLastName on the ComplexName object. (Converting a ComplexName into a SimpleName would result in losing the additional information stored in the ComplexName. For this reason, the original object, in this case the ComplexName, should be stored also.)

2.4 Summary

The DP system is implemented in Java, a class/instance object-oriented language. Information in the DP is represented as objects. These objects exist within a hierarchical system of classes and interfaces, which we call the kind/interface/implementation hierarchy. This hierarchy, along with Java’s reflection capabilities, gives the DP system a structured, yet flexible, object model.

Objects within a DP are organized based on the notion of containment. An object
can contain another object by either pointing to it or by storing its value in a field (such as in a compound object). The object which represents the DP itself contains all of the objects which represent a person's information.

Information objects descend from the DPObj class. DPObj provides many useful methods for manipulating these objects. The functionalities which are offered by these methods include accessing and setting the values of contained objects, learning about the object's place in the kind/interface/implementation hierarchy, and converting the object to a different type. Progressive revelation relies heavily on methods which are defined in DPObj. Many of these methods make use of the Java Reflection API.
Chapter 3

DP Ontology

3.1 The Need For a Default Ontology

Chapter 2 described how a DP structure can be extended and modified. Objects can contain an arbitrary number of other objects. In turn, the contained objects can be of arbitrary types. At some point, however, there should be some default objects which programmers can use. That way, many simple tasks can be done easily, without the programmer always having to create new kinds, interfaces, and implementations from scratch.

The objects described in the default ontology\(^1\) should be useful to programmers. Otherwise, no one will use the default ontology and everyone will end up writing their own objects anyway.

3.1.1 A Starting Point

We created our own ontology which we used to test our DP implementation. Our ontology was based largely on the data elements described in the P3P (Platform for Privacy Preferences Project) Syntax Specification [25]. The goal of P3P is to "enable Web sites to express their privacy practices and enable users to exercise preferences

\(^1\)For the remainder of the paper, the word ontology will be used to represent the specifications of the kind/interface/implementation hierarchies of a DP object and its inner objects. Since interfaces define containment structures, an ontology also defines a DP's internal structure.
over those practices." [24] Some of the P3P data elements in [25] include User, Date, and PersonName. The value of a data element can be either a string or another element. Our ontology also supports address books and shopping lists, which are not part of the P3P specification.

Since the P3P system specification does not indicate how a user's personal information should be stored, a user's information could be stored in a text file. We decided to store each data element as its own object. We then placed the P3P User object (representative of the person whose data was being kept) inside the "owner" field of the DP object.

3.2 Our Ontology and its Representation

After an ontology has been designed, a convenient representation for it must be found. What the suitable representation will look like depends on how complicated the ontology is. Regardless of the ontology, we know that one possible representation is the collection of Java code for the interfaces and classes that represent kinds, interfaces, and implementations. However, writing Java code is not always a compact way to convey information. Plus, as systems grow, the number of object types they contain will increase also, thereby increasing the amount of code needed to represent the ontology. If not much information is present in the code, then a more compact representation can be found.

Our Java code conveyed very little information. This was because our implementations were very basic: they did not have methods which performed computations or accessed external data sources.
3.2.1 Our Representation

A Tree-like Structure

We chose to represent our entire ontology using two tree-like structures. Within each tree are subtrees which represent kind/interface/implementation information. Each subtree has a height of at least four. The four levels correspond to the object's kind, interface, and implementation (2 levels). Inheritance in the object system, if there is any, will cause a subtree height greater than four. Regardless of the overall subtree height, each path from root to leaf can be divided into four sections, which correspond to the levels mentioned previously.

The subtree structure can be thought of as an expansion of what was shown in Figure 2-1. From top to bottom, the basic subtree levels are: kind name, interface name(s), implementation name(s), and object(s) (properties) pointed to by the implementation.

All subtrees have at least four levels, but not all of them have branchings that are greater than one. Branchings occur in the following places for the following reasons:

Between a kind node and an interface node There are multiple interfaces which belong to this kind.

Between an interface node and an implementation node There are multiple implementations which support this interface.

Between an implementation node and a property node There are multiple objects contained in this implementation.

Subtrees will have more than four levels if the kind has interfaces or implementations which inherit from other interfaces or implementations, respectively. In this case, descendant implementations will list only those properties which are new, not those which it has inherited from another implementation. See the Section 2.1.2 for

---

2For now, we will refer to them as trees. The reasons why they are not quite trees will be discussed soon.
more details. Either way, an object's tree will still have four distinct parts to it, even if corresponding parts are not always found at the same depth. These parts correspond to kinds, interfaces, implementations, and properties. For the following examples, we will separate tree parts (different from tree levels) by dotted lines.

First, let's look at a basic example. Figure 3-1 shows a single-branching tree associated with the kind CreditCard. (This is based on an example in Section 2.1.2). The tree has one kind (CreditCard), one interface (CreditCard1), one implementation (CreditCard1Impl), and one property (a String named accountNumber).

A more complicated example is shown in Figure 3-2, which illustrates the PhoneNumber kind. (This is also based on an example in Section 2.1.2.) The tree has one kind (PhoneNumber), two interfaces (PhoneNumberBasic and PhoneNumberAdvanced), two implementations (PhoneNumberBasicImpl and PhoneNumberAd-
vancedImpl), and two sets of two properties (two Strings named areaCode and number and two Strings named intCode and ext). PhoneNumberAdvanced and PhoneNumberAdvancedImpl inherit from PhoneNumberBasic and PhoneNumberBasicImpl, respectively.

These subtrees are not really trees because implementations can support more than one interface. Also, interfaces can extend one or more interfaces. If an ontology features multiple interface inheritance, then the tree which represents the ontology will have at least one cycle. Cycles are also created when an implementation extends another implementation and implements an interface, as in Figure 3-2.\textsuperscript{3} An example of a subtree with a cycle is given in Figure 3-3, which shows an implementation, Student1Impl, of kind Person which supports both the Student1 interface and the Adult1 interface. (The objects contained by the Student1Impl have been left off for clarity.) In reality, a subtree is actually a simple graph. Simple graphs are undirected and have no multiple edges or loops, but they can have cycles.

Figure 3-4 shows a complicated subtree featuring an implementation inheriting multiple interfaces, interfaces extending other interfaces, and implementations extending other implementations.

Once we had these kind trees, we divided them into two groups based on whether they were RealLifeObjects or DataObjects. This resulted in two trees. An example is shown in Figure 3-5. (The details of the kind trees were left out for clarity.) The structures in Figure 3-5 (combined with the kind trees) are our representation of an ontology.

An XML Document

We have found it useful to have a representation of our ontology using text. We used XML\textsuperscript{[22]} (Extensible Markup Language) as a shorthand way to express our trees. XML is a metalanguage that lets you design your own customized markup language. XML is written in SGML. An XML document looks similar to an HTML document

\textsuperscript{3}This type of cycle is mainly a by-product of our representation and does not involve multiple inheritance.
Figure 3-2: Hierarchy for kind PhoneNumber
in that they both have tags and sometimes there is data stored between tag pairs. The main difference is that the HTML tags are fixed, whereas the XML tags are arbitrary, based on what the author wants to accomplish. Thus, here we use XML to describe a DP ontology using just text. No graphical object modelling is necessary.

The XML specification of an ontology can be thought of as a textual representation of the tree in Figure 3-5. A pair of tags is used for each node in a tree. The data between the tags is everything below that node in the tree. If a node contains a subtree, then there will be more tags embedded between the node’s tags. When there is no data stored between a pair of tags, one tag may be used instead. This tag will be the regular tag name, appended by “/”.

For example, the XML specification of a book (including title, author, and the fact that it is in print) might look like this:

---

While the XML specification does not match the tree exactly, its structure is similar.
Figure 3-4: Complicated kind subtree
Let's apply these concepts to our kind/interface/implementaton hierarchy. Since the XML specification is hierarchical (like the tree it represents), we will start at the root (kinds) and work our way down to implementations and properties. Each explanation will discuss what types of data can be found between the tags mentioned in the list title.

Kind (<KIND>)

- The kind's name (<NAME>).

- Interfaces belonging to that kind (<INTERFACE>).

Interface (<INTERFACE>)

- The interface's name (<NAME>).
- Properties (contained objects) belonging to that interface (<PROPERTY>).
- Implementations belonging to that interface (<IMPLEMENTATION>).
- Interfaces extending that interface (<INTERFACE>).

Property (<PROPERTY>)

- The type (String, Person, etc.) of the object (<TYPE>).
- The fieldname of the object (<NAME>).
- Whether the object is a DP Vector (<VECTOR/>). (See Section 3.2.2 for more information.)

Implementation (<IMPLEMENTATION>)

- The implementation’s name (<NAME>).
- Any code that is to be added to the Java code for the implementation (<CODE>).
  (The need for this additional code will be explained in Section 4.2.)

The XML specifications for each kind/interface/implementation hierarchy are then divided into two groups, based on whether the kinds are RealLifeObjects or DataObjects. Each group of XML hierarchy specifications is surrounded by the <OBJECTTYPE> tags. Within the pair of <OBJECTTYPE> tags is also the name of the object type (<NAME>), which is either RealLifeObject or DataObject.

Finally, the two groups are combined into one XML document by surrounding them with the <ONTOLGY> tags. Each of the following examples shows the complete, stand-alone XML specification of a particular kind/interface/implementation hierarchy.

Examples

This is the XML specification for figure 4, the CreditCard example. Notice how the indented lines mimic the underlying tree structure.
<ONTOMETRY>
  <OBJECTTTYPE>
    <NAME>DataObject</NAME>
    <KIND>
      <NAME>CreditCard</NAME>
      <INTERFACE>
        <NAME>CreditCard1</NAME>
        <PROPERTY>
          <TYPE>String</TYPE>
          <NAME>accountNumber</NAME>
        </PROPERTY>
      </INTERFACE>
      <IMPLEMENTATION>
        <NAME>CreditCard1Impl</NAME>
      </IMPLEMENTATION>
    </INTERFACE>
  </KIND>
</OBJECTTTYPE>
</ONTOMETRY>

Here is the XML specification for figure 5, the PhoneNumber example. Notice how the interface inheritance is handled. Since the PhoneNumberAdvanced (PNA) interface extends the PhoneNumberBasic (PNB) interface, the XML specification for the PNA interface and its associated implementation (PhoneNumberAdvancedImpl) is located within the PNB interface’s <INTERFACE> tags.

<ONTOMETRY>
  <OBJECTTTYPE>
    <NAME>DataObject</NAME>
    <KIND>
      <NAME>PhoneNumber</NAME>
      <INTERFACE>
        <NAME>PhoneNumberBasic</NAME>
        <PROPERTY>
          <TYPE>String</TYPE>
          <NAME>areaCode</NAME>
        </PROPERTY>
        <PROPERTY>
          <TYPE>String</TYPE>
          <NAME>number</NAME>
        </PROPERTY>
      </INTERFACE>
      <IMPLEMENTATION>
        
      </IMPLEMENTATION>
    </KIND>
</OBJECTTTYPE>
</ONTOMETRY>
3.2.2 Features of Our Ontology and Its XML Representation

While working with our ontology, we discovered the need for objects which would hold several objects of the same kind or type. For example, an address book object would hold a collection of Person objects. To implement this idea, we created a data structure that combines the benefits of an array (enforcing object type/kind homogeneity) and a Vector (increasing data structure size as needed). In other words, our structure, which we call a DP Vector, is a type-parameterized Vector. A general system for parameterized types in Java, called PolyJ [17], has been developed at the MIT Laboratory for Computer Science.

A DP Vector can be contained within a DP object. In the XML representation of our ontology, a single <VECTOR/> tag designates a DP Vector. When a <VECTOR/> tag appears within a <PROPERTY> tag, then the contained object is a DP Vector which holds objects of the type or kind which appears between the <TYPE> tags. For example, the following XML code represents a contained object (property) which is a DP Vector that holds objects of the kind Person. The field name for the object is
3.2. OUR ONTOLOGY AND ITS REPRESENTATION

"entries."

<PROPERTY>
  <VECTOR/>
  <TYPE>Person</TYPE>
  <NAME>entries</NAME>
</PROPERTY>

Another feature of our XML representation and Java code generator is the ability to include specific pieces of code in the generated files. Code which is surrounded by the <CODE> tags (inside the <IMPLEMENTATION> tags) is automatically appended to the implementation class’ Java code. We used the <CODE> tag to give each DP object a special way of representing itself in HTML. The code was used in our DP browser, which will be presented in Section 4.2.

You can find the complete XML specification of our ontology in Appendix A.

3.2.3 Revising Our Ontology

While we experimented with our DP system, our ontology was always in a state of flux. Whenever we wanted to create new kinds, interfaces, or implementations, we had to write more Java code. We knew that there had to be a better way to handle changes in our ontology.

We wrote a Java program to take an XML ontology file as an input and automatically create all of the Java files needed to describe the kinds, interfaces, and implementations. This task would have been very difficult and time-consuming, had it not been for the existence of IBM’s XML4Java parser [11]. The XML4Java parser helps programmers parse, generate, and manipulate XML documents. We used the parser to parse the XML file into a tree based on the tags. We were then able to traverse the tree and use the data elements to create our Java files.

This solution made it relatively cheap to modify our ontology. The low cost encouraged us to go through more design revisions. Even after the initial design process has ended, automatic generation will continue to be convenient as our information storage needs change over time.
3.3 Designing a Truly Useful Ontology

While our ontology served our purposes well, we knew that designing a truly useful ontology would require a lot of effort. To make an ontology useful to a wide range of programmers, it should be designed by a group of people from different companies and research environments. Even if the agreed-upon ontology does not work for everyone, all is not lost. New objects can be created to meet specific needs, just as we created objects to represent address books and shopping lists. This will be discussed further in Section 3.3.1.

3.3.1 Beyond the Default Ontology (Flexibility and Extensibility)

There are many ways in which programmers can extend (or even replace) an ontology to meet their special needs. We have already discussed one way: by creating new interfaces and implementations. These interfaces and implementations would fall under one of the kinds established by the system. The programmer could either create entirely new interfaces and implementations or have them extend those which already exist, depending on the situation.

If the programmer does not like the kind system as it stands, he could also invent his own kinds. By this point, he has moved entirely beyond the existing ontology. However, the objects he creates can still descend from DPObj and reap the benefits of the kind/interface/implementation view of the world.

Using the DP's existing kind structure, instead of creating new kinds, would give a program useful information about a new interface or implementation. Reusing kinds, when appropriate, should be fairly easy, since kinds come with no functionality themselves.
3.4 Summary

An ontology is a specification of the kind/interfaceimplemenation hierarchies of a DP object and its inner objects. Each DP should come with a default ontology which has been implemented. Programmers will be able to use these default interfaces and classes as a starting point, rather than having to code new objects from scratch. Our default ontology is based on the data elements which are described in the P3P (Platform for Privacy Preferences Project) Syntax Specification.

The best way to represent a particular ontology (e.g., graphically or textually) depends on the complexity of the ontology. We envision our ontology as a large, tree-like structure. Each kind, along with its associated interfaces and implementations, is a subtree in the larger structure. The concrete representation of our ontology is an XML (Extended Markup Language) document. To help us experiment with classes present in our default ontology, we wrote a program which creates Java classes based on an XML ontology specification.

The most important aspect of a default ontology is that it be useful to programmers. In order for this to happen, people from a variety of commercial and research environments should work together to design the default ontology's kinds, interfaces, and implementations. When necessary, programmers can always extend the default ontology by creating new kinds, interfaces, and implementations.
Chapter 4

Interacting With a DP

The purpose of a DP is to make information useful across devices and applications. The next step in this process is deciding how people and computers should interact with a DP to obtain this information. This chapter covers interaction topics such as networking, security, and a graphical user interface for people to interact with a DP.

4.1 Computer Interaction

4.1.1 Networking

A DP is designed to be a free-running application with network accessibility. Computers and PIAs should be able to interact with a DP whether it is located in a different process or in an entirely different computer. While direct interaction with a DP occurs via the DP API, we have not yet specified how these API calls will be sent. In the current DP implementation, all personal information is contained within one object. This object, which represents the entire DP, must be loaded in by a program before the program can execute any API calls. Eventually, the DP must be separated from the PIAs which use it.

One way to transform the DP into a stand-alone entity is to use Java Remote Method Invocation (RMI). RMI allows applications to use objects easily which are
Figure 4-1: Network architecture for using a DP via RMI

located on remote hosts.¹ RMI, which is included in Java 1.1, works by giving applications a reference to the remote object. Using this reference makes the remote object seem as if it were in the same virtual machine as the application. This means that the application can invoke methods on the remote objects. Arguments which are passed to the method and values which are returned are all real objects.

Since a DP is an object, applications can use it via RMI. Once an application has a reference to a DP, it can invoke methods on the DP (such as DP API calls) by using RMI. The network architecture which would be used in this situation is shown in Figure 4-1. First, the DP registers itself and its location with the RMI Server. Then, an RMI Client contacts the RMI Server to request a reference to the DP. Once the reference has been received, the RMI Client can communicate with the DP via the RMI Server.

Creating remote objects is not difficult. The first step is to have the remote object extend the java.rmi.server.UnicastRemoteObject class, just as local objects extend the java.lang.Object class. The remote object must then implement a re-

¹RMI can also be used with objects which are located on the same machine.
remote interface. This remote interface extends the java.rmi.Remote interface. The methods in the remote interface must declare that they can throw the exception java.rmi.RemoteException. For more information on RMI, see [18].

The one drawback to RMI is that it handles only Java objects. This means that to communicate with the DP via RMI, each PIA would have to be running Java. The solution to this problem is to design a network protocol which would embody the DP API. This protocol would be used for communications between a client (PIA) and a server which is attached to a DP. The protocol would be similar to those which are used for HTTP and P3P. Designing the protocol would be time-consuming but not especially difficult. No matter how a client chooses to communicate with a DP, nothing can happen until the client is told where the DP can be reached (i.e., the location of the server which is attached to the DP).

Now that we have discussed how to implement the interaction between a PIA and a DP, we will illustrate this interaction by giving two short scenarios. The scenarios involve a PalmPilot and a cell phone which want to retrieve a phone number from a DP. The first scenario involves the PalmPilot and demonstrates a straight-forward interaction. The second scenario shows what happens when a cell phone encounters an unfamiliar datatype. In each case, the DP can be contacted via either RMI or a specialized DP network protocol.

**PalmPilot Scenario**

- The PalmPilot contacts the DP and queries it for a person’s phone number.

- The PalmPilot receives the phone number as an object and tests the object to see if it supports the PalmPilotPhoneNumber interface.

- The object does support that interface. The PalmPilot calls
  
  `{PalmPilotPhoneNumber}getPhoneNumber` and
  
  `{PalmPilotPhoneNumber}getAreaCode` and stores this information in the address book entry for that person.
Cell Phone Scenario

- The phone contacts the DP and queries it for a person's phone number.

- The phone receives the phone number as an object and tests the object to see if it supports the GenericCellPhonePhoneNumber interface.

- The object does not support that interface. The phone queries the object to find out whether it can be converted to a type which does support that interface.

- The object does not have any conversion functions other than toString. The phone gives up and calls toString on the object. It stores this information in the phone list entry for that person.

4.1.2 Security and Privacy

Both security and privacy are important issues to consider when designing a computer system. A system which contains sensitive information and is network-accessible, such as a DP, must address both security and privacy for its implementation to be complete.

Security and privacy are related, but it is important to understand that security is primarily a technical phenomenon, while privacy is a social phenomenon. Security deals with how to ensure that only certain people can manipulate pieces of information in specific ways. Privacy, on the other hand, covers a person's preferences about who can access which information and what can be done with the information after it has been given.

Security

The way in which a DP controls access to its information and the integrity of the information itself will depend on the owner's security preferences. Any security system, however, should address authentication, authorization, and encryption.
4.1. COMPUTER Interaction

Authentication Deciding who can access a piece of information is useless if users can masquerade as other users. This problem of determining who someone really is is known as authentication. For example, when a user logs in, the computer must decide whether he is who he says he is. Each server which is attached to a DP should also be authenticated. Otherwise, a program might send sensitive information to the wrong DP. Authentication can be accomplished in a variety of ways. Most of these methods are based on identifying what a user knows (e.g., a password), what a user has (e.g., a digital certificate), or what a user is (e.g., a certain fingerprint).

Authorization Once users have been correctly identified, it will be useful for the system to specify who is allowed to perform certain tasks. Deciding whether or not a person is allowed to do something is known as the authorization problem. Ways to make this decision fall into two categories, based on where the permissions information is located. The first category involves giving each piece of information a list of who is allowed to access it. Examples of this strategy are access control lists (ACLs). The other category involves giving each user a list of information which he is allowed to access. Examples of this strategy, also known as capabilities, are Kerberos tickets.

Encryption Lastly, a system must address the problem of encryption. Encryption can be used to prevent people from viewing information or from changing it without being detected. These problems can occur whether the information in question is stored in the DP or whether it is being transmitted over a network. The type of encryption which should be used (e.g., public key or private key) depends on the details of the situation.

One security technology which fits in well with the DP system is the Secure Sockets Layer Handshake Protocol [10], commonly referred to as SSL. SSL, which was developed by Netscape, supports server and client authentication and maintains the security and integrity of the transmission channel by using encryption and authentication.
Privacy

The DP owner's privacy preferences cover, for each piece of information, 1) who is allowed to access the information, 2) what each person is allowed to do with the information (e.g., read it and/or change it), and 3) under what circumstances the specified permissions apply. Remember that privacy is mainly a social phenomenon. One way to protect a person's privacy is to make both parties sign a contract before any information is revealed. If someone later breaks the agreement by misusing information, legal recourse can often be found.

One problem with privacy preferences is that there is no universally accepted vocabulary for expressing them. We suggest that the privacy mechanism of the DP system be something similar to P3P [24] (Platform for Privacy Preferences Project) and APPEL [23] (A P3P Preference Exchange Language). APPEL allows users to express their preferences in a set of preference rules. P3P can compare an outside request for information against a person's preference rules and either accept the request, reject it, or send back a counter proposal. Together, P3P and APPEL can help the DP decide whether a piece of information should be released or not.

4.2 Human Interaction

The applications can either interact with a DP directly via the DP API or indirectly via RMI or an HTTP-like protocol. These headless interfaces are suitable for applications which need to access DP information. If this information is to be useful to people, however, a DP should have a graphical user interface.

4.2.1 The DP Browser

We decided that the most convenient way for people to access DP information was through a web browser. We wrote an application which lives on a web server and contains a DP. The application, which we call a DP browser, gives people a graphical
user interface which they can use to access and change DP information.\(^2\)

**Using the DP Browser**

The browser interface consists of three panes. The top two panes are the largest. The one on the left has a tree view of objects in the DP containment hierarchy, with the DP itself as the outermost object. The one on the right has a list view of the objects which are contained by the tree view selection. Each pane refers to objects by the fieldnames in which they are stored by the enclosing object. For example, assume that a particular object stores a String in a field called "name". When the String object is mentioned in either the left or right pane, it will be referred to as "name". The third panel, which is underneath the other two, holds buttons which are used to accomplish certain tasks.

The user navigates by clicking on objects in either the left pane or the right pane. Once an object has been selected, two things will happen: 1) the tree view will update itself to represent the object's location in the DP containment hierarchy and 2) the list view will update itself to show the objects contained in the selected object. The list view shows detailed information for basic Java objects only (e.g., Strings or Integers). Since DP objects (those objects which inherit from DPObj ect) are more complicated, the list view provides a link to them instead of showing all the objects which they contain. For example, assume that a user has clicked on an object which has two fields: "description", which holds a String, and "friend", which holds an object of the kind Person. The right pane would display the contents of the friend String and a link to the Person object. Figure 4-2 shows how the browser looks when a user is browsing information. The selected object contains both DP objects and basic Java objects.

The DP browser can also be used to edit information. The bottom pane holds

\(^2\)The DP browser's implementation limits the user to working with a predefined DP ontology. In other words, a person cannot use a DP browser to create new kinds, interfaces, and implementations. This limitation is not a fundamental problem and can be overcome. One way to do this is to have the browser translate user actions into an XML ontology specification and then generate and compile Java code automatically.
Digital Persona Browser for Sabra Keln

<table>
<thead>
<tr>
<th>DP Object Hierarchy</th>
<th>Object Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>name:</td>
</tr>
<tr>
<td>birthdate</td>
<td>street: 123 Main St.</td>
</tr>
<tr>
<td>gender</td>
<td>city: Nowheresville</td>
</tr>
<tr>
<td>employment</td>
<td>stateProv: Califonia</td>
</tr>
<tr>
<td>home</td>
<td>postalCode: 12345</td>
</tr>
<tr>
<td>streetAddress</td>
<td>countryCode: US</td>
</tr>
<tr>
<td>name</td>
<td>country: USA</td>
</tr>
<tr>
<td>street</td>
<td></td>
</tr>
<tr>
<td>city</td>
<td></td>
</tr>
<tr>
<td>stateProv</td>
<td></td>
</tr>
<tr>
<td>postalCode</td>
<td></td>
</tr>
<tr>
<td>countryCode</td>
<td></td>
</tr>
<tr>
<td>country</td>
<td></td>
</tr>
<tr>
<td>telecomInfo</td>
<td></td>
</tr>
<tr>
<td>onlineInfo</td>
<td></td>
</tr>
<tr>
<td>business</td>
<td></td>
</tr>
<tr>
<td>dpUrl</td>
<td></td>
</tr>
<tr>
<td>cert</td>
<td></td>
</tr>
<tr>
<td>billTo</td>
<td></td>
</tr>
<tr>
<td>shipTo</td>
<td></td>
</tr>
<tr>
<td>dpFile</td>
<td></td>
</tr>
<tr>
<td>addressBook</td>
<td></td>
</tr>
<tr>
<td>shoppingList</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-2: Viewing information using the DP browser.
a button which says “Change Info”. When the user clicks on the button, the right pane displays a form. This form allows the user to change the information which is contained in an object (the object which is currently selected). Basic Java object information, such as strings and numbers, can be entered into the form. For DP objects, links are provided which take the user to the form appropriate for that object. The form fields appear with the current information already filled in. At this point, the bottom pane holds a button which says “Save to DP”. Clicking on this button saves the form’s information to the DP. Figure 4-3 shows how the browser looks when a user is editing information. The selected object contains only basic Java objects.

How the DP Browser Works

The DP browser displays information in HTML form. The HTML pages are created dynamically based on what information is stored in a DP. A web browser is used to display these pages. User actions in the browser environment are translated into DP
API calls, which are then sent to the DP. In this way, the DP browser exists at one level of abstraction above the computer-DP API level.

The method toHtmlTable produces the HTML code which displays information in the browser’s right pane. This method is desired for all information objects, since it is one of DPObjec’s helper methods. In the XML representation of our ontology, we used the <CODE> tags (inside of the <IMPLEMENTATION> tags) to redefine the behavior of toHtmlTable in a few implementation classes. Adding this code allowed us to display different objects in different ways.

The key technology in the DP browser is WBI [1] (Web Browser Intelligence). WBI is a programmable HTTP request processor which stands as in intermediary between the web browser and the DP. The DP browser works by running WBI as a server. When the user requests some information via the web browser, WBI intercepts the HTTP request and translates it to a DP API call. WBI then sends the call to the DP.\(^3\) By generating HTTP responses also, WBI can dynamically create HTML pages.

### 4.3 Summary

Both people and computers should be able to access information through a DP. The simplest way for a computer to interact with a remote DP is through Java RMI (remote method invocation). This solution will work only if the client computer is running Java. If this is not the case, then a network protocol (similar to HTTP) should be designed which embodies the DP API. Security and privacy issues should be addressed in a complete implementation of a DP system. Some of the technologies which might be useful here include SSL (Secure Sockets Layer), P3P (Platform for Privacy Preferences Project), and APPEL (A P3P Preference Exchange Language).

The easiest way for a person to interact with a DP is through a graphical user interface. We created an application, called a DP browser, which lives on a web server.

\(^3\)Currently, WBI loads in the DP when it starts running and writes changes to it on exit. Once the network protocols have been worked out, the DP will exist apart from WBI.
and gives people access to a DP via a web browser. A DP browser can be used to query and edit information. The browser works by translating user actions into DP API calls. These calls are then sent to the DP.
Chapter 5

Conclusion

5.1 Conclusion

The DP system exhibits a unique balance between structure and flexibility. This balance was made possible by the use of diverse technologies such as objects, interfaces, and reflection. The structure and flexibility of a DP make the task of managing a set of personal information much easier. A DP makes it possible to interact with all of a person's information by using only one centralized protocol, regardless of how the information is represented or where it is physically located. This aspect of the DP system can help solve information management problems such as synchronization and aggregation. However, the usefulness of the current DP implementation merely hints at the power that such a system could possess. More work must be done before the DP system can reach maximum effectiveness: the object model should be refined; the utility of the default ontology should be proven; a network protocol should be designed; and security and privacy should be implemented.

5.2 Future Work

There are many possibilities for future work involving the Digital Persona system. Some of these research areas were discussed in Chapter 4, such as implementing network capabilities via RMI or a specialized protocol, security systems, and privacy
preferences. Other areas include writing user-level applications which interact with the DP, such as the DP browser, and helper applications which aid PIAs and DPs in synchronizing information and performing other housekeeping tasks.

The largest project involving the DP system is determining how useful it is in practice. The DP’s unique object model, reflective architecture, and default ontology need to be used in real-world applications. Fully implementing a DP and then testing it will most likely uncover some problem areas and opportunities for improvement, as well as some untapped possibilities. One area in particular which interests us is the usefulness of distinguishing between real life objects (RLOs) and data objects (DOs). Although the current DP system treats RLOs and DOs in the same way, they are fundamentally different in real life. These differences should find their way into the DP system in terms of object implementations. Some issues to think about in this area are:

- The unary properties of DOs are constant, while the unary properties of RLOs can change. For example, whether a person is happy or not can change over time. The number associated with a particular credit card cannot change over time. This idea can be extended to say that unary properties of DOs are immutable. If the information stored in a DO changes, then the DO is fundamentally not the same object anymore. For example, each phone number (set of 10 digits) is unique and immutable. If the area code field of a phone number object is changed, then the object is no longer the same object.

- Containment via compound objects makes sense for some DOs. For example, an address object can physically contain a zip code object. Compound objects involving RLOs, especially people, is more difficult. In real life, people are connected by different types of relationships (friends, relatives, coworkers). No one person is “above” another person or “contains” another person. When people are modelled as objects, they should all exist on the same “level.” An object representing a manager should not contain (via compound objects) objects which represent employees. Instead, the manager object should somehow point
to or be linked to the employee objects. This idea boils down to the different types of containment (as mentioned in Section 1.3.2) which should be used for RLOs versus DOs.

- Some objects exist solely to group other objects. For example, objects representing address books and committees group people objects. These “grouping” objects are neither RLOs nor DOs. Perhaps a third category should be created for them with its own special implementation issues.
Appendix A

XML Ontology of Our DP

```xml
<?xml version="1.0"?>
<ONTOMETRY>

<OBJECTTYPE>
   <NAME>DataObject</NAME>
   ...

<KIND>
   <NAME>AddressBook</NAME>
   <INTERFACE>
      <NAME>AddressBook1</NAME>
      ...

<VARIABLE/>
   <TYPE>Person</TYPE>
   <NAME>entries</NAME>
   ...

<IMPLEMENTATION>
   <NAME>AddressBook1Impl</NAME>

<CODE>
<![CDATA[
public String toHtmlTable(String path) {
   /* Returns html code for a listing of Persons (first and last names), which
      are links. Clicking on a Person link will take you to the browser
      for that Person. This method overrides the default toHtmlTable
      method as defined in DObject. */
   ...
   System.out.println("in toHtmlTable for address book");
   StringBuffer htmlSb = new StringBuffer();
   htmlSb.append("<TABLE BORDER=0 CELLPADDING=3>");
   Object fieldValue = new Object();
   AddressBook1 addbk = (AddressBook1) this;
   System.out.println("length of addbk: " + addbk.getLength());
   Object elt;
   for (int i = 0; i < addbk.getLength(); i++) {
      ...
```
elt = addbk.addElement(i);
htmlSb.append("<TR><TD>");
htmlSb.append(makeLink(path + "/" + i));
PersonName1 pn = (PersonName1)((Person1)elt).getName();
htmlSb.append(pn.getFirstName() + " " + pn.getLastName() + "</A>");
htmlSb.append("</TD></TR>");
} // closes for loop
htmlSb.append("</TABLE>");
return htmlSb.toString();
}]]>
</CODE>
</IMPLEMENTATION>
</INTERFACE>
</KIND>

<KIND>
<NAME>ShoppingList</NAME>
</INTERFACE>
<PROPERTY>
<NAME>ShoppingList1</NAME>
</PROPERTY>
</VECTOR/>
<TYPE>Commodity</TYPE>
<NAME>items</NAME>
</PROPERTY>

<PROPERTY>
<TYPE>String</TYPE>
<NAME>listOfVendors</NAME>
</PROPERTY>
<IMPLEMENTATION>
<NAME>ShoppingList1Impl</NAME>
</IMPLEMENTATION>

<CODE>
 <! [CDATA[
 public String toHtmlTable(String path) {
    /* Returns html code for a listing of commodity item descriptions, which
    are links. Clicking on a commodity link will take you to the browser
    for that commodity. This method overrides the default toHtmlTable
    method as defined in DPObjec. */
    
    System.out.println("in toHtmlTable for shopping list");
    StringBuffer htmlSb = new StringBuffer();
    htmlSb.append("<TABLE BORDER=0 CELLPADDING=3>");
    Object fieldValue = new Object();
    ShoppingList1 sl = (ShoppingList1)this;
    System.out.println("length of shplist: " + sl.getLength());
    Object elt;
    for (int i = 0; i < sl.getLength(); i++) {
        elt = sl.getElementById(i);
        htmlSb.append("<TR><TD>");
        htmlSb.append(makeLink(path + "/" + i));
        htmlSb.append(((SimpleCommodity)elt).getItemDescription() + "</A>");
    } /* else { // elt is a Person */
    PersonName1 pn = (PersonName1)((Person1)elt).getName();
    htmlSb.append(pn.getFirstName() + " " + pn.getLastName() + "</A>");
} ]]>
} */
htmlSb.append("</TD></TR>"
);  
} // closes for loop
htmlSb.append("</TABLE>");
return htmlSb.toString();
}
]]>
</CODE>
</INTERFACE>
</KIND>

<KIND>
  <NAME>ID</NAME>
</INTERFACE>
  <NAME>P3PID</NAME>
  <PROPERTY>
    <TYPE[String]</TYPE>
    <NAME>PUID</NAME>
  </PROPERTY>
  <PROPERTY>
    <TYPE[String]</TYPE>
    <NAME>TUID</NAME>
  </PROPERTY>
</IMPLEMENTATION>
  <NAME>P3PIDImpl</NAME>
</IMPLEMENTATION>
</INTERFACE>
</KIND>

<KIND>
  <NAME>JobInfo</NAME>
</INTERFACE>
  <NAME>JobInfo1</NAME>
  <PROPERTY>
    <TYPE[String]</TYPE>
    <NAME>employer</NAME>
  </PROPERTY>
  <PROPERTY>
    <TYPE[String]</TYPE>
    <NAME>department</NAME>
  </PROPERTY>
  <PROPERTY>
    <TYPE[String]</TYPE>
    <NAME>jobTitle</NAME>
  </PROPERTY>
  <PROPERTY>
    <TYPE[String]</TYPE>
    <NAME>employeeNumber</NAME>
  </PROPERTY>
</IMPLEMENTATION>
  <NAME>JobInfo1Impl</NAME>
</IMPLEMENTATION>
</INTERFACE>
<KIND>
  <NAME>PostalAddress</NAME>
  <INTERFACE>
    <NAME>PostalAddress1</NAME>
    <PROPERTY>
      <TYPE>PersonName</TYPE>
      <NAME>name</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>street</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>city</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>stateProv</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>postalCode</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>countryCode</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>country</NAME>
    </PROPERTY>
  </INTERFACE>
</KIND>

<KIND>
  <NAME>TelecomInfo</NAME>
  <INTERFACE>
    <NAME>TelecomInfo1</NAME>
    <PROPERTY>
      <TYPE>PhoneNumber</TYPE>
      <NAME>phone</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>PhoneNumber</TYPE>
      <NAME>fax</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>PhoneNumber</TYPE>
    </PROPERTY>
  </INTERFACE>
</KIND>
<NAME>mobile</NAME>
</PROPERTY>
<PROPERTY>
  <TYPE>PhoneNumber</TYPE>
  <NAME>pager</NAME>
</PROPERTY>
<IMPLEMENTATION>
  <NAME>TelecomInfo1Impl</NAME>
</IMPLEMENTATION>
</INTERFACE>
</KIND>

<KIND>
  <NAME>PhoneNumber</NAME>
  <INTERFACE>
    <NAME>PhoneNumber1</NAME>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>intCode</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>locCode</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>number</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>ext</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>comment</NAME>
    </PROPERTY>
  </INTERFACE>
</KIND>

<KIND>
  <NAME>OnlineInfo</NAME>
  <INTERFACE>
    <NAME>OnlineInfo1</NAME>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>email</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>URI</NAME>
    </PROPERTY>
  </INTERFACE>
</KIND>
<IMPLEMENTATION>
  <NAME>OnlineInfoImpl</NAME>
</IMPLEMENTATION>
</INTERFACE>
</KIND>

<KIND>
  <NAME>ContactInfo</NAME>
  <INTERFACE>
    <NAME>ContactInfo1</NAME>
    <PROPERTY>
      <TYPE>PostalAddress</TYPE>
      <NAME>streetAddress</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>TelecomInfo</TYPE>
      <NAME>telecomInfo</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>OnlineInfo</TYPE>
      <NAME>onlineInfo</NAME>
    </PROPERTY>
    <IMPLEMENTATION>
      <NAME>ContactInfoImpl</NAME>
    </IMPLEMENTATION>
  </INTERFACE>
</KIND>

<KIND>
  <NAME>Date</NAME>
  <INTERFACE>
    <NAME>Date1</NAME>
    <PROPERTY>
      <TYPE>Integer</TYPE>
      <NAME>year</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>Integer</TYPE>
      <NAME>month</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>Integer</TYPE>
      <NAME>day</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>Integer</TYPE>
      <NAME>hour</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>Integer</TYPE>
      <NAME>minute</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>Integer</TYPE>
    </PROPERTY>
  </INTERFACE>
</KIND>
<KIND>
  <NAME>Person</NAME>
  <INTERFACE>
    <NAME>Person1</NAME>
    <PROPERTY>
      <TYPE>PersonName</TYPE>
      <NAME>name</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>Date</TYPE>
      <NAME>birthdate</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>gender</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>JobInfo</TYPE>
      <NAME>employment</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>ContactInfo</TYPE>
      <NAME>home</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>ContactInfo</TYPE>
      <NAME>business</NAME>
    </PROPERTY>
    <PROPERTY>
      <TYPE>String</TYPE>
      <NAME>dpUrl</NAME>
    </PROPERTY>
  </INTERFACE>
  <IMPLEMENTATION>
    <NAME>Person1Impl</NAME>
    <INTERFACE>
      <NAME>P3PUser</NAME>
      <PROPERTY>
        <TYPE>String</TYPE>
        <NAME>cert</NAME>
      </PROPERTY>
      <PROPERTY>
        <TYPE>ContactInfo</TYPE>
        <NAME>billTo</NAME>
      </PROPERTY>
      <PROPERTY>
        <TYPE>ContactInfo</TYPE>
        <NAME>shipTo</NAME>
      </PROPERTY>
      <PROPERTY>
        <TYPE>String</TYPE>
        <NAME>dpFile</NAME>
      </PROPERTY>
    </INTERFACE>
    <IMPLEMENTATION>
<NAME>P3PUserImpl</NAME>
</IMPLEMENTATION>
</INTERFACE>
</IMPLEMENTATION>
</INTERFACE>
</KIND>

<KIND>
<NAME>Commodity</NAME>
<INTERFACE>
<NAME>SimpleCommodity</NAME>
<PROPERTY>
<TYPE>String</TYPE>
<NAME>itemDescription</NAME>
</PROPERTY>
<PROPERTY>
<TYPE>String</TYPE>
<NAME>manufacturer</NAME>
</PROPERTY>
<PROPERTY>
<TYPE>Integer</TYPE>
<NAME>askingPrice</NAME>
</PROPERTY>
<PROPERTY>
<TYPE>String</TYPE>
<NAME>additionalComments</NAME>
</PROPERTY>
<PROPERTY>
<TYPE>String</TYPE>
<NAME>contactInformation</NAME>
</PROPERTY>
<PROPERTY>
<TYPE>String</TYPE>
<NAME>howToContactBidder</NAME>
</PROPERTY>
</IMPLEMENTATION>
<NAME>SimpleCommodityImpl</NAME>
<INTERFACE>
<NAME>ComplexCommodity</NAME>
<PROPERTY>
<TYPE>String</TYPE>
<NAME>warranty</NAME>
</PROPERTY>
<PROPERTY>
<TYPE>String</TYPE>
<NAME>model</NAME>
</PROPERTY>
<PROPERTY>
<TYPE>String</TYPE>
<NAME>condition</NAME>
</PROPERTY>
<PROPERTY>
<TYPE>String</TYPE>
<NAME>features</NAME>
</PROPERTY>
<IMPLEMENTATION>
  <NAME>ComplexCommodityImpl</NAME>
</IMPLEMENTATION>
</INTERFACE>
</IMPLEMENTATION>
</INTERFACE>
</KIND>

</OBJECTTYPE>

</ONTOMETRY>
Bibliography


THESIS PROCESSING SLIP

FIXED FIELD: ill. name

index biblio

● COPIES: Archives Aero Dewey Eng Hum

Lindgren Music Rotch Science

TITLE VARIES: □

NAME VARIES: ✓ Rachel

IMPRINT: (COPYRIGHT)

COLLATION: 103 p

ADD: DEGREE: □ DEPT.: □

SUPERVISORS:

NOTES:

cat': date:

DEPT: E.E. page: F.39


NAME: KELIN, Sabra-Anne R.