Working with Java Interfaces and Classes

How to separate interfaces from implementations

By Dirk Riehle and Erica Dubach

Software engineering has been using interfaces for over 25 years. Java, in contrast to other object-oriented programming languages like C++ and Smalltalk, provides a clean separation between interfaces and classes that implement the interfaces. For new Java developers, this requires some adjustment in thinking. This article shows not only how to distinguish between interfaces and classes, but also how to use them effectively to model and implement Java-based systems. We illustrate the concepts using the implementation of a simple CORBA-based naming service.

1 Introduction

Java provides developers with both classes and interfaces for the construction of software systems. The distinction between classes and interfaces is a well-established and undoubtedly important concept in software engineering. However, the effective use of interfaces in Java is not always intuitive, as several examples from Sun's JDK demonstrate.

This article describes the importance of interfaces and the distinction between interfaces and classes for Java-based application development. It demonstrates the interaction between the concept of interface, abstract class and concrete class, and shows how they work together in helping you implement software systems and applications that are more flexible and are more easily maintained than would be possible without the proper use of these concepts.

The article uses the simple and frequently found example of a naming service to illustrate the concepts. The naming service definition uses the CORBA Object Services Specification (COSS) [1].

2 Classes and Interfaces in Design and Implementation

The object-oriented paradigm is applicable to the analysis, design and implementation of software systems. Different aspects of the various object-oriented concepts come into play during the different phases.

In software engineering, the term 'object' is used in two different senses: in the Anglo-Saxon tradition, the term 'object' is usually defined as the encapsulation of data and operations. In the Scandinavian tradition, an object is a phenomenon encountered in a specific application domain. There are similar differences for the term 'class': for some, a class is a blue-print used to create objects, for others it is the abstraction of similar objects found in some application domain.

These definitions obviously complement each other, where one stems from the technical domain of design and implementation and the other from the application-specific analysis and design domain. Good methodologies point to this distinction and allow for these concepts and their specific applications to co-exist harmoniously.

Copyright 1999 Dirk Riehle and Erica Dubach. All rights reserved.
Published in Java Report 4, 7 (July 1999). Page 35-46.

The Scandinavian school of thinking that produced the object-oriented programming languages SIMULA 67 and Beta has put forth a continuous effort to use the same concepts to describe analysis, design and implementation and to come up with one notation to fit all three phases. SIMULA 67 can be regarded as the world's first object-oriented language, which had a significant influence on subsequent object-oriented programming languages. The effects of the effort to create one notation can be seen in modern programming languages, such as C++ and Smalltalk, whose main object-oriented programming concept is the class concept.

In Java, an additional concept is added to the one of classes, namely interfaces. Though interfaces are frequently mentioned in the context of both C++ and Smalltalk, interfaces could not be separated from a class or from their implementation. Java now permits the definition of an interface without requiring its implementation, a feat that can be accomplished in C++ only by defining abstract classes with 'pure virtual' methods.

Interfaces define the access to a given object, allowing that object's implementation to be changed without forcing the clients that are using the interface to change their implementation. Java interfaces group many related methods without providing information about their implementation. Java interfaces can be created using multiple inheritance, while classes in Java use single inheritance.

Java lets developers create new interfaces by inheriting from several instead of just one interface. This feature allows an object to be accessed from several different views, i.e., from several different interfaces. As an example, an interface can be defined through inheritance from several interface (much like protocols [2]), each of which might describe technical or application domain specific aspects that might not be useful on their own.

Interfaces in Java are implemented using classes, which is indicated by the 'implements' relationship. Instances of classes that implement a specific interface may be used in all situations where a client expects objects conforming to the interface.

The introduction of interfaces in Java programming is an important improvement over other programming languages in which the distinction between classes and interfaces is not specifically made. However, it might also be confusing, since most analysis methods still only uses classes. As an example from the banking domain, the concepts 'person' and 'account' are defined as classes during analysis (or even design), but in the Java implementation, these would be interfaces! This means that concepts, defined as classes during analysis, are frequently expressed as Java interfaces on the implementation level. Java classes are then created to implement these interfaces.

The following examples illustrate the consequences of the distinction between classes and interfaces for the design and implementation of Java-based systems. Specifically, several design and implementation patterns are described, which demonstrate how to design with interfaces. They then use abstract classes to improve code reuse.

3 Interfaces and Implementation

The following example is used throughout the article and shows the design and implementation of a naming service as defined in the OMG CORBA Object Services Specification (OMG-COSS) [1]. The purpose of a naming service is to name an object for later retrieval. The operating principle is "Object for Name". This section describes the design and implementation of names with interfaces and concrete classes.

Examples of names are the URL "http://www.ubs.com/ubilab" or the file name "net/projects/Geo/Vroom". Names might appear to be a simple concept at first glance, to be represented by a string. In reality a string is only of limited use for names that have a complex internal structure. The interpretation of such names is best represented by the definition of a Name interface.

The following Java interface describes such a name interface:

```
public interface Name
{
  public Name context();
  public Name without(Name name);
  public Name extended(String nc);
  public Name context(int end);
  public Name subname(int start, int end);

  public Vector components();
  public int noOfComponents();
  public String component(int i);
  public String lastComponent();

  public boolean contains(Name name);
}
```

A name consists of an ordered list of name components, in accordance with the OMG-COSS definition. This means that the name “net/projects/Geo/Vroom” consists of the four components “net”, “projects”, “Geo” and “Vroom”. A separator is usually employed to display the name as a single string, such as the “/” used for file names in Unix. For simplicity, this article equates name components with strings.

Given this definition, a name object can be queried for a Vector of name components (public Vector components()). Further, the context name of a name object can be retrieved, which for our example of “net/projects/Geo/Vroom” will result in “net/projects/Geo”, and the closing name object requested, which returns “Vroom”, etc.

To implement the Name interface, a class needs to be defined. There are different possible implementations, which will help demonstrate the power of distinguishing between classes and interfaces.

For one, a name can be implemented as a Vector of name components, i.e., strings. This would be done using the following class, VectorNameImpl:

```
public class VectorNameImpl
  implements Name
{

// implementation state
protected Vector fComponents;

public VectorNameImpl()
{
  fComponents = new Vector();
}

public VectorNameImpl(Vector components)
{
  fComponents = components;
}

public Name context()
{
  Vector components = fComponents.clone();
  components.removeElementAt(components.size()-1);
  return new VectorNameImpl(components);
}

public String lastComponent()
{
  return (String) fComponents.lastElement();
}

... etc.

}
```

Using a Vector of strings as the Name implementation facilitates the efficient return of name parts as a new name, or just returning the last component of a name. However, the effort of implementing a concept that appears superficially simple with a Vector may seem excessive for some applications, which leads to the following alternative implementation.

Just as a name object can be implemented using a Vector, one can imagine using a single string to represent the name. The string contains all name components, each adjacent to the next, distinguished from each other with the designated separator. Within each name component of the string representing the full name, the separator must be masked. This results in the class StringNameImpl, which also implements the Name interface from which objects can be created, as was the case for the VectorNameImpl class.

```
public StringNameImpl
   implements Name
{

// implementation state
protected String fNameString;

public StringNameImpl()
{
   fNameString = new String();
}

public StringNameImpl(String nameString)
{
   fNameString = maskSeparator(nameString);
}

public Name context()
{
   char sepChar = separatorChar();
   int index = fNameString.lastIndexOf(sepChar);
   String contextString = fNameString.substring(0, index-1);
   return new StringNameImpl(contextString);
}

public String lastComponent()
{
   char sepChar = separatorChar();
   int index = fNameString.lastIndexOf(sepChar);
   return fNameString.substring(index, fNameString.length());
}

protected char separatorChar()
{
   return ‘/’;
}

protected String maskSeparator(String inputString)
{
   // mask separator in inputString...
}

... etc.

}
```

Both the StringNameImpl and the VectorNameImpl class provide an implementation of the same interface and can be exchanged as needed, with no need for clients to change their implementation.

Guidelines are required to allow a name object to be exchanged during run-time, independent of the classes that implement it. Such a guideline was outlined in the Gang-of-four book “Design Patterns: Elements of Reusable Object-Oriented Software”:

Program to an interface, not an implementation. ([3], page 18)

This means that only the name of an interface should be used, in our example: Name. The names of the two classes StringNameImpl and VectorNameImpl are used only to create the objects. This guideline is also the reason why our projects use the postfix "Impl" for classes, instead of other versions sometimes found, such as "I" or "Ifc" for interface. Most code should be written so that it refers to interfaces only, which is why we prefer to tag the implementation classes rather than the interfaces.

Figure 1 shows the interface and its two implementations.

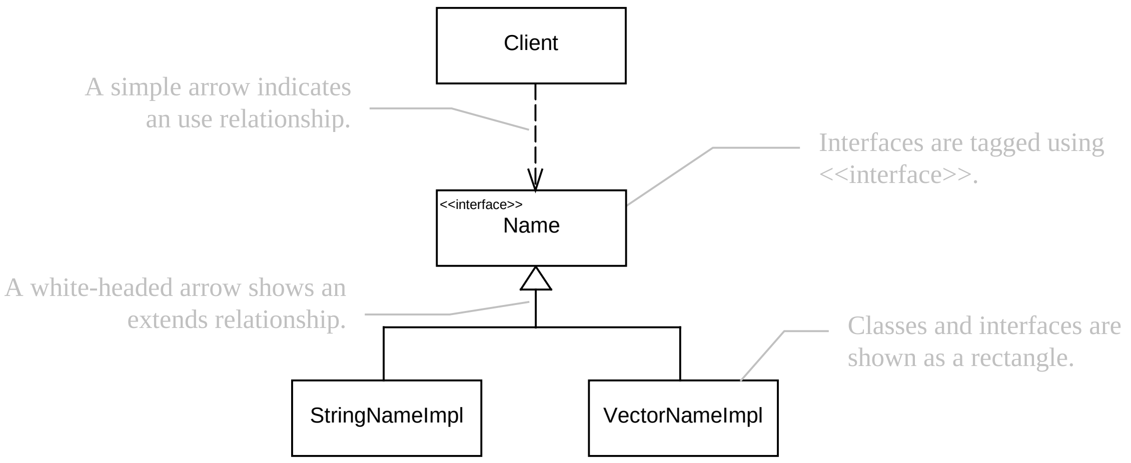
Client
A simple arrow indicates an use relationship.
Interfaces are tagged using <<interface>>.
<<interface>>
Name
A white-headed arrow shows an extends relationship.
Classes and interfaces are shown as a rectangle.
StringNameImpl
VectorNameImpl

Figure 1: The Name interface with the two implementation classes StringNameImpl and VectorNameImpl.

The client code does not depend on the class name, if clients use the interfaces implemented by the class. The concrete class only needs to be named for instantiating the object. However, even this remaining dependency can be cut, using the appropriate design patterns. For example, this can be accomplished with the patterns "Abstract Factory" [3, page 87] or "Product Trader" [4, page 29]. The effect of this would be that the name of the implementing classes is no longer part of the client code, making it independent from them. As an added benefit, the implementations can be extended and changed without affecting the client code.

The next question that poses itself is which criteria to use in choosing a specific implementation. There is no unqualified answer, since it depends on the intended applications. For the naming service example, the VectorNameImpl class is preferable because name components are used frequently, and efficient access to them is important. In contrast, StringNameImpl would be best used when a large number of names will be used and efficiency in storage is crucial.

Through this discussion it becomes obvious that differentiating between the implementations is still important. However, the differences in the implementations and their effects on client code can usually be minimized as long as these differences are only relevant during the creation of the objects, and the client code remains unaffected by them.

4 Abstract and Concrete Classes

The last section described the use of interfaces and their simple implementation with concrete classes. In order to maximize re-use, thought needs to be given to how the classes should inherit from each other. This leads to thinking about the concepts of abstract and concrete classes, which this section deals with.

On a general modeling level, a class is an abstract class if there are no concrete instances of it. As an example, the class “LifeForm” is an abstract class since no instances of it exists. However, there are concrete subclasses, such as “Person” or “Cat”, of which instances may exist. This in turn assumes that “Person” and “Cat” do not themselves have further subclasses, which would make them abstract classes themselves.

On the programming level, a class is understood to be abstract when its implementation is not complete, but rather represents scaffolding for its subclasses. Abstract classes are methodically employed as the scaffolding that is then filled by the concrete subclasses, which complete the implementation. As detailed below, this is done using the protected interfaces provided within the class inheritance hierarchy. Since the implementation of an abstract class is incomplete, there are no instances of an abstract class.

Abstract classes, as defined by Java, are those that are marked by the keyword “abstract”, which results in them not being instantiable. A similar effect is achieved when no public constructors are offered. A class is considered concrete if it can be instantiated, i.e., objects exists. Within Java, therefore, concrete classes cannot be declared as abstract and must have at least one public constructor (or an equivalent method like static factory methods).

Back to our example: of course, the definition of Name objects does not provide sufficient functionality to store objects under a name or to retrieve them again. The actual naming service is still missing. Fortunately, the naming service interface is defined by OMG-COSS, which we simply adopt:

```
public interface NamingContext
{
  public void bind(Name name, Object object);
  public void rebind(Name name, Object object);
  public void unbind(Name name);
  public Object resolve(Name name);
  public boolean contains(Name name);
}
```

This interface enables a client to store objects under a certain name (bind(Name,Object)) and then to retrieve them using that name (resolve(Name)). Objects can also be renamed (rebind(Name,Object)), as well as removed (unbind(Name)). Finally, contains(Name) requests whether a certain object is known by the naming service.

The following code stores a new object (such as a browser) under a certain name (such as “Customer-Browser”) with the naming service.

```
Browser browser = new BrowserImpl();
Name browserName = new StringNameImpl(“CustomerBrowser”);
LocalServices.getNamingService().bind(browserName, browser);
...
```

At a later point, the following code can be used to retrieve the browser object using the name “Customer-Browser”:

```
Name browserName = new StringNameImpl(“CustomerBrowser”);
Object object = LocalServices.getNamingService).resolve(browserName);
Browser browser = (Browser) object;
...
```

As can be seen by inspecting the last line of code, using a naming service defined in this way, objects are retrieved only via the interface Object and type checking must be performed by the client code. Here however, downcasting to “Browser” should never fail.

The next question to be answered is how to implement NamingContext. The central part of the implementation must obviously be a mapping from Name to Object (or to be precise, to object references). The specific implementation of this mapping can take on drastically different forms. The most obvious and simple implementation uses hash-tables for an associative lookup of the object reference to a given name. This works best when all names and objects are to be held in memory, and where this is possible, given the expected volume and capacity. If this is not the case, an alternative mapping implementation should be chosen.

Some CORBA NamingService implementations maintain the name/object mapping in files, for example. One reason for this is that the mappings must be accessible after re-starting the naming service, without loss of information. Looking up an object reference in a file-based representation of retrieving the corresponding object reference must obviously operate with different mechanisms than a simple lookup in a hash-table.

The memory-resident implementation is fast, yet is feasible only up to medium-sized object volumes. The file version is slow, especially if no caching is employed, but safe and suitable for large object volumes. Once again, this shows that there is no better or worse implementation, only trade-offs that must be evaluated for the implementation of a given task.

What is the simplest way to deal with the different implementations, and thereby maximize code reuse? The solution lies in the appropriate use of abstract classes, which implement the shared aspects of the different implementation classes, but leave open the differences.

The class hierarchy for our naming service example is shown in Figure 2 below.

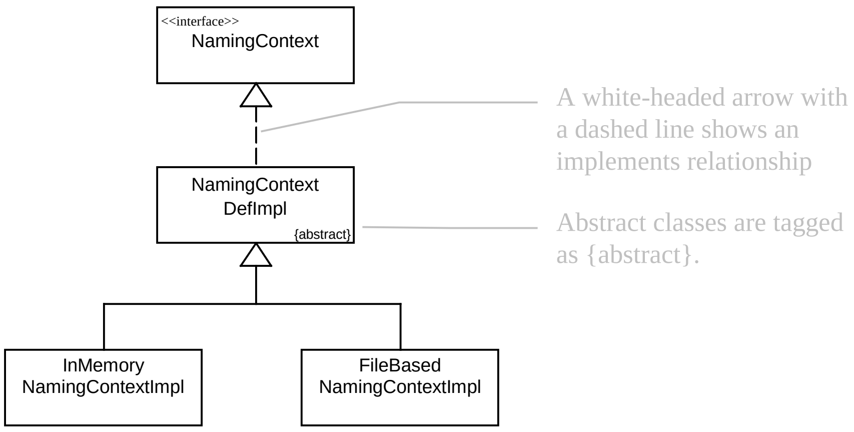
<<interface>>
NamingContext
A white-headed arrow with a dashed line shows an implements relationship
NamingContext DefImpl
{abstract}
Abstract classes are tagged as {abstract}.
InMemory NamingContextImpl
FileBased NamingContextImpl

Figure 2: The interface for NamingContext with two implementation classes, whose shared features are defined in the abstract superclass NamingContextDefImpl.

The abstract class NamingContextDefImpl implements the shared aspects of its intended subclasses (DefImpl stands for Default Implementation). This is achieved, for example, by using the design patterns "Factory Method" ([3], page 107) and "Template Method" ([3], page 325). This approach is commonly known as the narrow inheritance interface principle. The code for NamingContextDefImpl is as follows:

```
public abstract class NamingContextDefImpl
  implements NamingContext
{

protected NamingContextDefImpl()
{
  // since this is an abstract class,
  // no public contructors may exist
}

public void bind(Name name, Object object)
{
  if ( ! contains(name) ) doBind(name, object);
}

public void rebind(Name name, Object object)
{
  if ( ! contains(name) ) return;
  doUnbind(name);
  doBind(name, object);
}

... and the other methods

// this is the inheritance interface
protected abstract void doBind(Name name, Object object);
protected abstract void doUnbind(Name name);
protected abstract void doContains(Name name);

// that is all! No implementation state which would
// unnecessarily complicate the subclasses

}
```

NamingContextDefImpl has been prepared to be the abstract superclass for the definition of future subclasses. At the same time some functionality has already been implemented, such as the public methods of NamingContext. The implementation is not complete however, since those parts that differ from each other for the subclasses are delegated to the implementations of those subclasses. In this example, the variations are the abstract methods doBind, doUnbind, doContains in the NamingContextDefImpl class.

These methods constitute the inheritance interface (sometimes also called the “protected” interface). This interface is only seen by subclasses, which constitute a different kind of clients than client classes that make use of the public methods of an object through its regular interfaces. The inheritance interface should be as small as possible (or “narrow”, hence the name “narrow inheritance interface principle”), and its methods should be as orthogonal as possible. Its primary purpose is to provide those primitive methods, based on which the regular public methods of the interfaces of the class can be implemented.

These methods can now be implemented by the subclasses in the specific way necessary. For example, InMemoryNamingContextImpl uses a hash-table:

```
public class InMemoryNamingContextImpl
  extends NamingContextDefImpl
{

// implementation state
protected Hashtable fNameTable;

public InMemoryContextImpl()
{
  // this is a concrete class and therefore
  // has a public constructor
  fNameTable = new Hashtable();
}

protected void doBind(Name name, Object object)
{
  fNameTable.put(name, object);
}

protected void doUnbind(Name name)
{
  fNameTable.remove(name);
}

protected boolean doContains(Name name)
{
  fNameTable.contains(name);
}

}
```

Of course the methods doBind, doUnbind and doContains are implemented differently in the FileBased-NamingContextImpl class, where they are based on file accessing and caching.

The art of successfully designing the abstract superclass NamingContextDefImpl consists of finding the inheritance interface usable by a large variety of subclasses, upon which NamingContextDefImpl can be implemented. This example demonstrates a successful abstraction that provides a high degree of code reuse for the subclasses, without losing flexibility.

Of course it is possible to conceive of a completely different implementation of NamingContext, in which NamingContextDefImpl cannot be reused. This however, is precisely the reason for distinguishing between classes and interfaces: in such a case this other implementation class might implement NamingContext directly (perhaps named TotallyUnrelatedNamingContextImpl), without inheriting from NamingContextDefImpl, since it cannot reuse any of NamingContextDefImpl's code.

5 Packaging

How do you package interfaces and classes? First of all, it is important to determine what clients of your code should see and what should be hidden from them. Typically, they should see the interfaces, but not the implementation classes.

As a consequence, interfaces and implementation classes should be provided as different packages. For our example, this leads to the following packages:

```
package com.ubs.ubilab.Naming; // provides interfaces
package com.ubs.ubilab.NamingImpl; // provides naming service implementation
package com.ubs.ubilab.NamingTest; // provides test suites for implementation
```

The separation of interfaces, implementations, and test code into different packages is a well-understood concept. New implementations might either be added to the NamingImpl package or be provided as another package next to existing implementation packages. For example, if the two different implementations we have discussed (in-memory and file-based management) grow to be sufficiently big, they might be separated and provided as two different implementation packages:

```
package com.ubs.ubilab.NamingImpl; // provides abstract classes
package com.ubs.ubilab.InMemoryNamingImpl; // provides in-memory impl.
package com.ubs.ubilab.FileBasedNamingImpl; // provides file-based impl.
```

An alternative to this structure is to make the implementation and test packages sub-packages of the interface package. This way, clients see only the super-ordinate interface package. For our example, this leads to the following package structure:

```
package com.ubs.ubilab.Naming; // provides interfaces
package com.ubs.ubilab.Naming.Impl; // provides implementations
package com.ubs.ubilab.Naming.Test; // provides test suites
```

We have long used the first variant, but have started using the second variant more frequently now. The problem with the first variant is that it clutters your name space with many packages of differing significance for users. Users are interested in the interface package, once in a while they are interested in the implementation package, and almost never they are interested in the test suites.

The second variant resolves this problem by hiding the less significant packages deeper in the hierarchy. Here, 'less significant' is to be understood from the user-developer's point of view - after all, that is our client who we design and implement technical concepts like naming services for.

Separating interfaces from implementation classes is all very fine, but still client code needs a way to create instances of classes. As explained, this might mean that clients nevertheless have to access the implementation classes.

One convenient workaround is to provide final static classes in the interface package that access the implementation package but return objects typed by the interfaces. Typically, the access methods of these classes should give high-level information about the behavior of the returned object, but without providing specific implementation class names.

For our example, we provide a class called NamingSap (Sap stands for “Single Access Point”) in the Naming package:

```
package com.ubs.ubilab.Naming; // interface package

public final class NamingSap
{
   // default implementation for those clients that
       //don’t care about the implementation details
   public final NamingContext newNamingContext()
   {
      return newFastNamingContext();
   }

   // fast but not too robust implementation
   public final NamingContext newFastNamingContext()
   {
      return new com.ubs.ubilab.InMemoryNamingContextImpl();
   }

   // slower but more robust implementation
   public final NamingContext newRobustNamingContext()
   {
      return new com.ubs.ubilab.FileBasedNamingContextImpl();
   }
}
```

This class introduces a convenient indirection that decouples client code from specific class names, but still provides information about the desired object behavior in terms of performance or memory consumption. Of course, this is only a simple scheme, and more elaborate object creation mechanisms can be employed, as mentioned in section 3.

6 Summary

This article shows how to differentiate between classes and interfaces in the modeling and implementing of Java-based systems and how to use them effectively. The concepts covered are the separation of interface and implementation, the use of abstract and concrete classes, and how to package them. These concepts are illustrated using a simple naming service.

We would like to thank our colleagues Om Damani, Frank Fröse, Erich Gamma, Zsolt Haag, and Kai-Uwe Mätzel for the discussions and their feedback on this article.

We further would like to thank Frances Paulisch and Michael Stal, who originally encouraged us to write this article for Java Spektrum, the German sister magazine of Java Report [5].

Bibliography

[1] OMG. *CORBA Object Services Specification.* Framingham, MA: Object Management Group, 1997.

[2] NeXTStep. Object-oriented Programming and the Objective C Language. NeXTStep developer’s library. NeXT Computer Inc., 1993.

[3] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] Dirk Bäumer and Dirk Riehle. „Product Trader“. *Pattern Languages of Program Design 3.* Addison Wesley, 1998. Chapter 3.

[5] Dirk Riehle. “Arbeiten mit Java-Schnittstellen und -Klassen”. *Java Spektrum* 5/97 (September/October 1997). Page 26-33.

About the authors

Dirk Riehle works as a software engineer at Credit Suisse in Zurich, Switzerland. He wrote this article while working at Ubilab, the IT innovation laboratory of UBS AG. Dirk is the author of many journal articles about object-oriented software development. He is also an editor of *Pattern Languages of Program Design 3,* the most recent volume of design patterns from the PLoP and EuroPLoP conference series. He can be contacted at riehle@acm.org.

Erica Dubach holds an MS of Software Engineering from Depaul University in Chicago, and wrote this article while working at Ubilab, the IT innovation laboratory of UBS AG. Currently she works in the New Technology department of Atraxis AG (SAir Group) in Zurich, Switzerland and can be contacted at dubach@acm.org.