Working with Java Interfaces and Classes

How to maximize design and code reuse in the face of inheritance

By Dirk Riehle and Erica Dubach

Software engineering has been using interfaces for over 25 years. The distinction between interfaces and implementations is an important issue of object-oriented software system design. Java supports this distinction, yet for many developers, in particular with a C++ or Smalltalk background, the proper use of interfaces may not be intuitive right from the beginning. This article briefly discusses the general distinction between interfaces and classes as interface implementations, and then describes how to deal with interface inheritance and how to factor implementations to maximize code reuse. The implementation of a simple CORBA-based naming service is used as an ongoing example.

1 Introduction

Java provides developers with both classes and interfaces for the construction of software systems. On the one hand, the distinction between interfaces and their implementations is a recognized and undisputedly important concept in software engineering. On the other hand, well-known examples, such as the Abstract-Window-Toolkit (AWT), demonstrate that the effective use of interfaces is not always intuitive.

This is the second in a series of two articles. The first article appeared in the July 1999 issue of Java Report. It discussed the importance of interfaces and the distinction between interfaces and the classes that implement them. Section 2 of this article briefly reiterates the topic to set the stage for more advanced issues of interface and class inheritance as well as code factoring. The example used throughout the article is the design of a naming service, as defined by the CORBA Object Services Specification (COSS) [1].

2 Interfaces and Implementation

Object-oriented modeling is based on classes and objects. A class is the abstraction of several similar phenomena, which are found in one or in more domains. Objects are the concrete phenomena themselves.

In Java-based implementations of software systems, classes defined in domain models are typically represented as Java interfaces. These interfaces in turn are implemented by additional Java classes.

It is also possible to represent a domain concept, defined as a class in the domain model, directly as a Java class, without the introduction of an interface. However, this solution has several drawbacks. The major issue is that the interface and implementation become one unit of code (i.e., the class) and clients directly depend on this particular class, which prevents exchanging or choosing it freely. Since systems are destined to evolve, this is likely to lead to problems sooner rather than later. Furthermore, any changes to the class directly effect the client code, which possibly requires re-coding or re-compiling. The use of interfaces helps decouple clients from the classes that implement a particular domain class.

Thus, distinguishing between interfaces and classes that implement them reduces problems of system change and evolution and can significantly increase flexibility, adaptability and maintainability.

Published in Java Report 4, 10 (October 1999). Page 34pp.

The first of the two articles uses the implementation of a naming service to illustrate these advantages. A name, such as “net/projects/Geo/Vroom/” is made up of several name components. Different approaches can be taken to implement the concept of name. The interface Name is defined and two example implementations are shown. In one, the Name interface is implemented using the class StringNameImpl, which maintains the names as a single string with masked separators (“/”). The other implementation is the class VectorNameImpl, which maintains the individual name components as strings in a Vector. Neither implementation can be considered superior to the other; the intended use by a client determines which implementation fits best.

The first article further describes the tasks of a naming service and discusses two possible implementations. A naming service attaches a name to an object, stores the object under that name in order to retrieve it later using the given name. The following code describes the naming service as the interface *NamingContext*, which was taken directly from the CORBA Object Services Specification for the naming service [1]:

```
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);
}
```

The methods bind() and resolve() serve to store and retrieve the objects under a certain name. Using the following code, a customer browser object can be stored and retrieved later:

```
// store under name CustomerBrowser
Browser browser = new BrowserImpl();
Name browserName = new StringNameImpl(“CustomerBrowser”);
LocalServices.getNameService().bind(browserName, browser);
...
// retrieve under name CustomerBrowser
Name browserName = new StringNameImpl(“CustomerBrowser”);
Object object = LocalServices.getNameService().resolve(browserName);
Browser browser = (Browser) object;
...
```

We introduced two different implementations of the NamingContext interface, both of which were represented as concrete classes that implement the interface. One is based on a hash-table that resides in memory, which maps names to object references. The other implementation maintains names and object references in one or more files. Storing the information in memory is fast and works well for a small and mid-sized numbers of object/name pairs, though the data is lost in the case of a system crash. The file-based implementation on the other hand is capable of surviving a crash, since each object/name pair is stored on the hard drive. This second solution is also more suited for larger numbers of objects, but it is usually slower than the memory-resident implementation, at least for the first access, if caching is used.

These two implementations are defined as the InMemoryNamingContextImpl and FileBasedNamingContextImpl classes. The abstract superclass NamingContextDefImpl captures their similarities. The resulting design is shown in Figure 1:

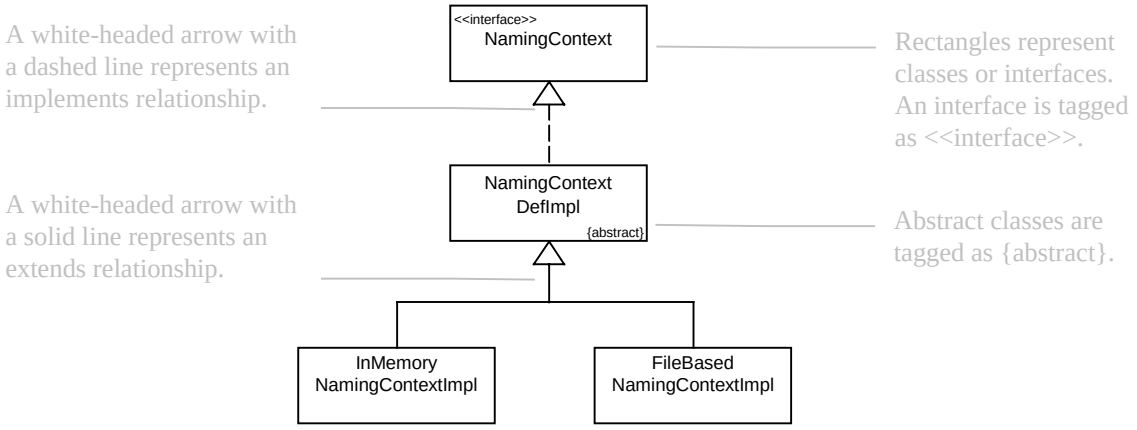

Figure 1: The example of the NamingContext interface with two concrete implementation classes and a common abstract superclass.

The abstract class NamingContextDefImpl captures common implementation aspects of the NamingContext interface and represents scaffolding for the subclasses. The scaffolding is filled out by these concrete subclasses. The abstract superclass implements the common aspects of its subclasses (DefImpl stands for default implementation) using the concepts of "Factory Method" ([2], page 107) and "Template Method" ([2], page 325). The definition of NamingContextDefImpl is as follows:

```
public abstract class NamingContextDefImpl
   implements NamingContext
{

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 further implementation state which would
// (possibly unnecessarily) burden the subclasses

}
```

The abstract superclass NamingContextDefImpl has been defined and can now be inherited from. Some functionality has already been implemented, such as the public methods of NamingContext, but not the functionality that needs to vary in the subclasses. The methods doBind, doUnbind and doContains define the functionality that needs to vary from subclass to subclass (they constitute the narrow inheritance interface). Therefore, they are represented as protected abstract methods of the class NamingContextDefImpl. All code in NamingContextDefImpl delegates subclass-specific functionality to these abstract methods, which are then implemented by concrete subclasses.

The methods doBind, doUnbind and doContains are implemented using a hash-table in the subclass InMemoryNamingContext, and using a file-based implementation for the FileBasedNamingContextImpl

class. The art of designing the abstract superclass NamingContextDefImpl consists of finding a suitable inheritance interface based on which NamingContextDefImpl can be implemented. Subclasses then only fill out this inheritance interface.

After this brief review, we now dive into more advanced concepts.

3 Interface and Class Inheritance

The previous sections show how to implement an interface and how to use abstract and concrete classes to improve code reuse. Software systems, however, consist of complex interface hierarchies, which leads us to examine the inheritance of interfaces and how they relate to the concept of implementation classes and code reuse.

To demonstrate this, we continue the naming service example. An object name, such as "http://www.riehle.org/SwissPhone1.html" is a complex construct, consisting of several parts, each of which requires elaborate lookup methods. A web browser first interprets the protocol to use, then the Internet Domain Name Service supplies the IP address for "www.riehle.org" and finally an http server delivers the content of "SwissPhone1.html" by reading a file for the given file name. The file name interpretation in the local file system might be complex as well, for example when symbolic links use NFS-Mount-Points to refer to files across file systems.

It is desirable to separate those parts of a name which require complex implementations for resolving them. Ideally, the handling of the separated parts can be changed without affecting the other parts.

The *Composite* design pattern ([2], page 163) tells us how to do so. It describes how to build recursive object hierarchies and is ideal for hierarchical name spaces [3]. A hierarchical file system is a good example, where each folder defines the name space for the files it contains. From each folder the files can be accessed without having to provide the full path name. Figure 2 shows a simple file tree.

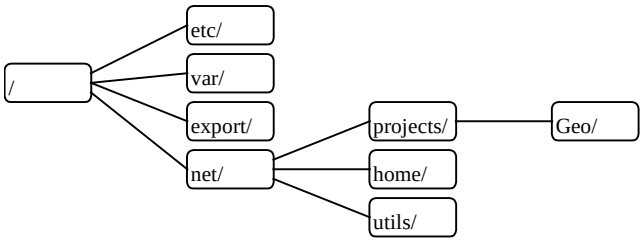

Figure 2: A file system as an example of a hierarchical name space.

The file named "/net/projects/Geo/Geo.proj" can be referenced from the so-called naming context "/" using the name "net/projects/Geo/Geo.proj", or from "/net/" using the name "projects/Geo/Geo.proj", etc. An object with a name ending with the "/" separator is a so-called naming context, which can take a given name and return the appropriate object, usually a file, directory or symbolic link. The interface for such naming contexts is defined as NamingContext above.

To resolve the name "net/projects/Geo/Geo.proj", it makes sense that the naming context "/" does not do all the work. Rather, "/" should know its sub-contexts and delegate the work to them, since they themselves are naming contexts. Sub-contexts of "/" from the above example, are "etc/", "var/", "export/" and "net/". To resolve the name of a file, the naming context first removes the part of the name that leads to the selected sub-context. Therefore, "/" removes "net/" from the name "net/projects/Geo/Geo.proj", leaving "projects/Geo/Geo.proj". The shortened name gets passed to the sub-context object, which in turn goes through the same process, until the searched-for file "Geo.proj" is found. A search down the tree structure is the result from the initial name.

In real applications, naming contexts do not map onto directories one-on-one, typically because of efficiency reasons. A reasonable solution, for example, represents each file system with a naming context object that is attached to an embedding super-ordinate naming context, where it is mounted.

To add sub-contexts to embedding context objects, the interface NamingContext needs to be extended. This leads to the interface *CompositeNamingContext*, which extends NamingContext:

```
public interface CompositeNamingContext
   extends NamingContext
{
   public void bindSubContext(Name name, NamingContext nc);
   public void rebindSubContext(Name name, NamingContext nc);
   public void unbindSubContext(Name name);
   public NamingContext resolveSubContext(Name name);
   public boolean containsSubContext(Name name);
   public Vector getSubContexts();
}
```

An object that implements the CompositeNamingContext interface is an object that first of all functions as a normal naming context object, since it inherits its definition from NamingContext. In addition, parts of the name space it is responsible for can be attached as sub-contexts. This is supported by methods for the management of child-objects in a tree structure, like getSubContext(), bindSubContext(), etc. The interface CompositeNamingContext is derived from the COSS specification for naming services [1].

A composite naming context object delegates the search for an object to one of its sub-contexts when the object name indicates that the sub-context is responsible for it.

An implementation for CompositeNamingContext must therefore provide additional functionality to the NamingContext implementation. First, it must be able to manage sub-contexts as part of its name space, and secondly, it must adjust the NamingContext functionality to function properly in this extended use scenario.

- The implementations of the methods bindSubContext, rebindSubContext, etc. are similar to the NamingContext methods bind, rebind, etc. Typically, the sub-context might be searched for in a memory-resident hash-table to see if it is part of the local name space. Sometimes, however, a file-based solution might be preferable, for similar reasons like the ones discussed above.
- Additionally, the functions bind, rebind, etc. must be adjusted. A check must be performed before each call to these functions as defined in NamingContext, to see if the name to be resolved is in the name space directly managed by the current naming context object, or whether it must be passed on to a sub-context.

Lets consider a few possible implementations before looking at specific code examples. The inheritance tree might be different depending on the kind of re-use desired.

Let us assume that the class InMemoryCompositeNamingContextImpl implements the memory-resident solution to manage the sub-contexts. Should the class then inherit from InMemoryNamingContextImpl? If so, the resulting implementation resolves the name as well as maintains the sub-contexts in memory.

On the other hand, if an implementation is chosen that manages names using a database and where the sub-contexts are in files, then the class inherits from FileBasedNamingContextImpl. A different combination needs to be chosen if the name table is to be kept in memory and the sub-contexts stored as files.

Concrete classes, such as InMemoryNamingContextImpl are “semantically closed”, meaning that their implementation determines their entire behavior. A further implementation consideration is that any subclass of such a semantically closed class is likely to contradict the inherited implementation, a phenomenon described as part of the abstract superclass rule [4].

A solution to this problem is presented below: it takes into account that inheriting from concrete classes is usually not a good idea since the implementation semantics might get confused and an explosion of implementation variations is produced.

Figure 3 shows the suggested class hierarchy that uses the idea of abstract default implementations introduced above.

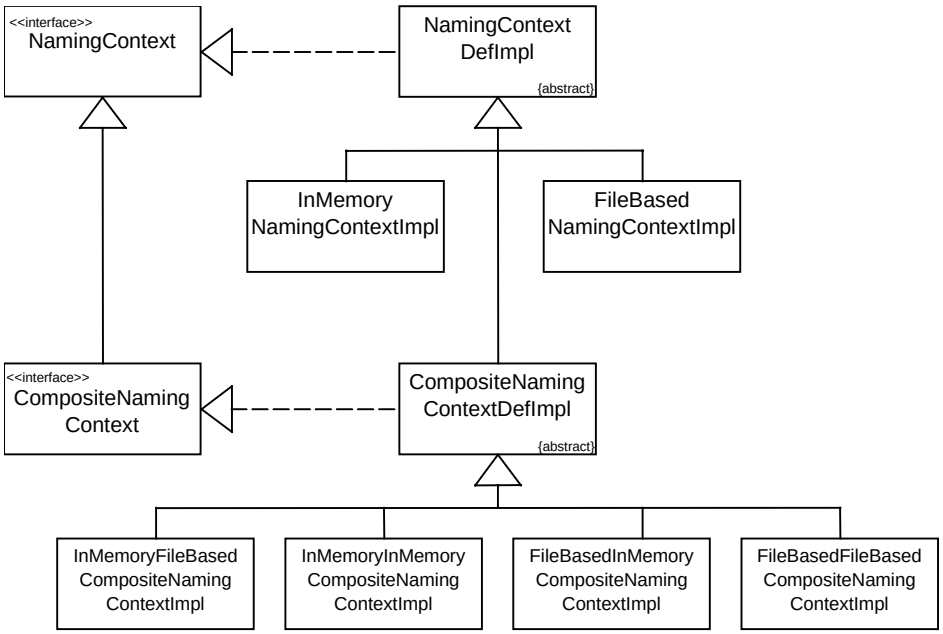

Figure 3: Illustration of interface and default implementation inheritance

Here again an abstract default implementation class exists with the name of CompositeNamingContextDefImpl, which inherits from NamingContextDefImpl. Just like the NamingContextDefImpl class, this implementation uses a narrow inheritance interface:

```
public abstract class CompositeNamingContextDefImpl
   extends NamingContextDefImpl
   implements CompositeNamingContext
{

... constructors and initialization code

public void bindSubContext(Name name, NamingContext nc)
{
   if ( ! contains(name) ) doBindSubContext(name, nc);
}

public void rebindSubContext(Name name, NamingContext nc)
{
   if ( ! contains(name) ) return;
   doUnbind(name);
   doBind(name, nc);
}

... and the other methods

// here the inheritance interface
protected abstract void doBindSubContext(Name name, NamingContext nc);
protected abstract void doUnbindSubContext(Name name);
protected abstract void doContainsSubContext(Name name);

}
```

All these functions are parallel to NamingContextDefImpl. The main part of the implementation is based on a small inherited interface, requiring the subclasses only to fill out the inherited interface.

The bind, rebind, etc. methods need to be adjusted, of course. This is done using a helper method that determines if a name is in the current name space or within a registered sub-context:

```
// belongs to CompositeNamingContextImpl
protected NamingContext findSubContext(Name name)
{
   for ( int i = 0; i < name.noOfComponents(); i++ )
   {
      Name context = name.context(i);
      if ( containsSubContext(context) )
      {
         return resolveSubContext(context);
      }
   }
   return null;
}
```

Using this helper method, the function resolve() can determine if the name to be resolved is maintained locally or whether the task needs to be passed to a sub-context:

```
// belongs to CompositeNamingContextDefImpl
public Object resolve(Name name)
{
   NamingContext subContext = findSubContext(name);
   if ( subContext != null )
   {
      Name subName = name.without(subContext.getName());
      return subContext.resolve(subName);
   }
   else
   {
      return super.resolve(name);
   }
}
```

The concrete subclasses of CompositeNamingContextDefImpl only need to implement the unfinished scaffolding parts. Those are the methods from the inheritance interface of NamingContextDefImpl for the name management and the inheritance interface of CompositeNamingContextDefImpl for the sub-context management. Since there are two possible implementations for each, four combinations of subclasses are possible, as is also visible in Figure 3. This combinatorial explosion can be counteracted using the implementation factoring technique detailed in the next section, though the principle is discussed here.

For each interface an abstract default implementation class exists that runs parallel to the interface hierarchy, but that is not instantiable. The more complex the hierarchy is, the more inheritance interfaces exist that must be implemented by the subclasses. By using the concept of default implementation a certain degree of direct code reuse can be achieved, which is particularly good in the face of the limitation in Java that classes can only inherit from a single superclass.

If the concept represented by an interface is to be instantiable, then concrete subclasses of the corresponding abstract default implementation class should exist that can be used directly by the clients. For pragmatic reasons this is useful because clients do not have to introduce a subclass themselves, and it is useful for psychological reasons since it reduces the complexity of using the design.

The next question to ask is what happens when the interface hierarchy is not based on simple inheritance but uses multiple inheritance. This is possible, of course, though the motivation for using multiple inheritance should be inspected first. From our experience, the domain-specific modeling usually produces a simple inheritance structure. Multiple inheritance is only useful for attaching additional protocols to a single-inheritance tree. These protocol interfaces are recognizable by their ending “–able”, e.g. Clone-

able, Serializable, Remote. They usually do not have their own default implementation, but if they do, the following concepts discussed in the next section are very helpful.

4 Factoring Implementations

As the previous section showed, complex objects often consist of different aspects that can easily be separated in the implementation. A composite naming context needs to manage its name space as well as its sub-contexts. These tasks both happen in the same object, yet the implementations of them are not necessarily dependent on each other.

The introduction of an abstract superclass, which implements the standard behavior of an interface leads to the concept of an inheritance interface. An abstract superlass defines a set of protected class-hierarchy internal methods, the inheritance interface, in such a way that the superclass' own implementation can make use of these methods without having to implement them. Figure 4 uses the ongoing example and shows the inheritance interface for the management of the name space and where it is implemented. The bold lines represent the definition of an inheritance interface in a class and a bold line with a box indicates the implementation.

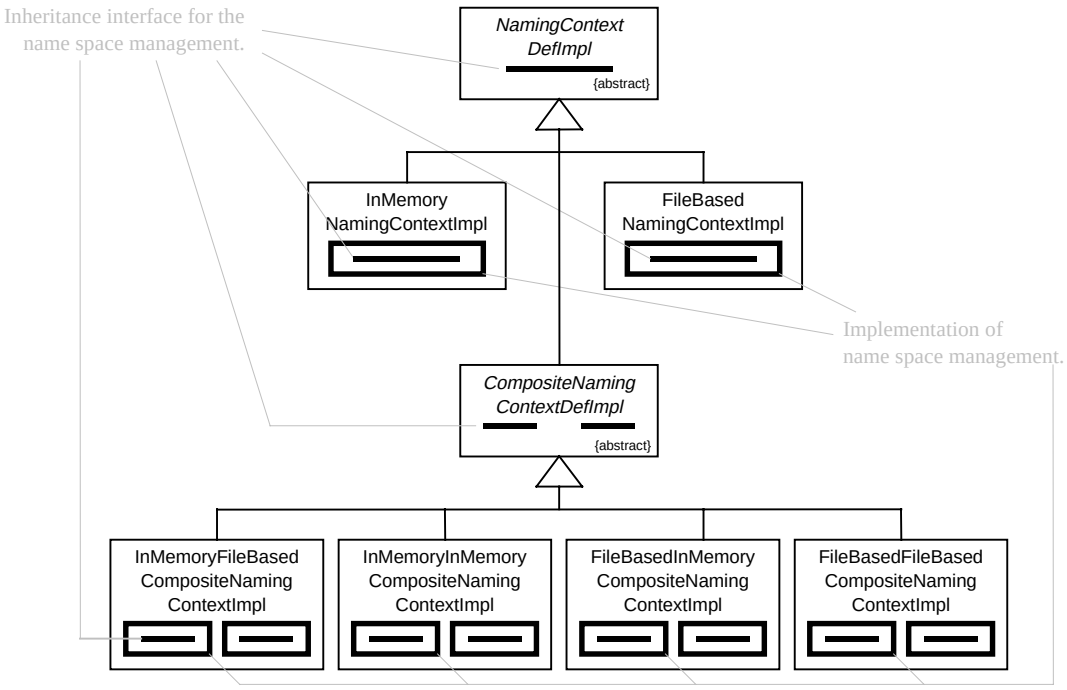

Figure 4: Illustration of the inheritance interfaces of abstract superclasses

The CompositeNamingContextDefImpl level adds the inheritance interface for the management of sub-contexts, which results in an explosion of possible subclasses. These are the classes InMemoryFileBased-, InMemoryInMemory-, FileBasedInMemory-, and FileBasedFileBasedcompositeNamingContextImpl. (Note: These horribly long names will be used throughout for the illustration of our example. For the implementation however, shorter names would be preferable or the long names could be hidden behind convenience methods).

The prefixes InMemory and FileBased indicate the chosen implementation for the management of the name space and the sub-contexts. The two classes InMemoryFileBased- and InMemoryInMemory-CompositeNamingContextImpl both implement an in-memory name space management, but use different mechanisms to implement the sub-context management. They cannot inherit from a common superclass

though (such as InMemoryNamingContextImpl), without losing other advantages. The main advantage in this example is the superclass CompositeNamingContextDefImpl, which defines the standard behavior for composite naming contexts. Of course, it is possible to choose a different superclass, though this would result in conflicts for other combinations, which is a result of the limitation that classes only have single inheritance. No matter how much effort is put into the design of a class hierarchy, optimal code reuse is not possible in Java using only inheritance.

How to avoid the implementation redundancy and the resulting hassles with maintenance and enhancements? Using object composition instead of inheritance for code reuse. The most important enabler for this is the concept of inheritance interfaces, which we have already introduced.

The implementation of the inheritance interface is passed to a new object, instead of implementing it in every subclass. This new object is an example of a class which implements this (and only this) interface. Since the objects might differ, the implementations can differ as well, as long they satisfy the inheritance interface. In our example, this leads to the following design:

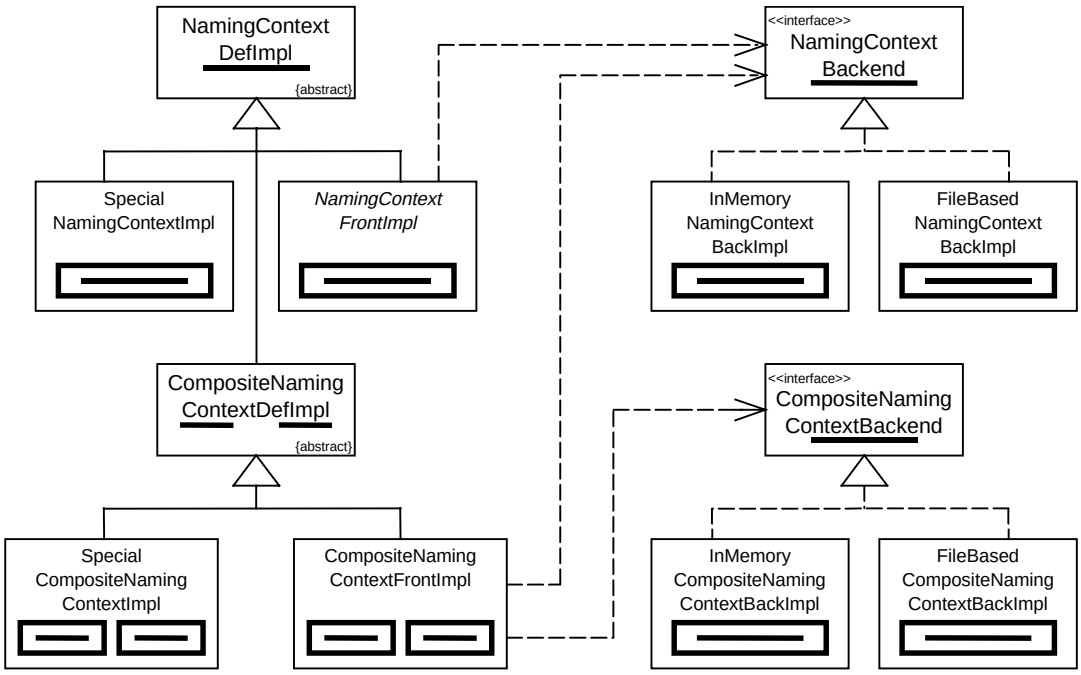

Figure 5: Parameterizing of implementation classes with implementation strategies.

The new subclass NamingContextFrontImpl of NamingContextDefImpl implements the inheritance interface by delegating the function calls to those objects that implement the interface NamingContextBackend. For our example, these could be the classes InMemoryNamingContextBackImpl or FileBasedNamingContextBackImpl.

Another class, CompositeNamingContextFrontImpl, delegates the implementation of the name space management and the sub-contexts to other objects. For each part of the implementation there is a “Backend” class which fulfills just this aspect and whose objects provide the implementation for the “Frontend” classes.

The implementation of the inheritance interface for NamingContextFrontImpl is as follows:

```
public class NamingContextFrontImpl
   extends NamingContextDefImpl
{

// implementation state
protected NamingContextBackend fNamespaceBackend;
```

```
public NamingContextFrontImpl(NamingContextBackend ncBackend)
{
   fNamespaceBackend = ncBackend;
}

protected void doBindSubContext(Name name, NamingContext namingContext)
{
   fNamespaceBackend.bindSubContext(name, namingContext);
}

protected void doUnbindSubContext(Name name);
{
   fNamespaceBackend.unbindSubContext(name)
}

protected boolean doContainsSubContext(Name name);
{
   return fNamespaceBackend.containsSubContext(name);
}

}
```

The following code produces a naming context object.

```
// selecting the implementation
NamingContextBackend ncBackend = new InMemoryNamingContextBackImpl();
// creating the naming context object
NamingContext nc = new NamingContextFrontImpl(ncBackend);
```

In the case of the composite naming context the code is similar, expect that it is parameterized with two implementation objects.

```
public class CompositeNamingContextFrontImpl
   extends CompositeNamingContextDefImpl
{

// implementation state
protected NamingContextBackend fNamespaceBackend;
protected CompositeNamingContextBackend fSubcontextBackend;

public CompositeNamingContextFrontImpl(NamingContextBackend ncBackend,
   CompositeNamingContextBackend cncBackend)
{
   initialize(ncBackend, cncBackend);
}

protected initialize(NamingContextBackend ncBackend,
   CompositeNamingContextBackend cncBackend)
{
   fNamespaceBackend = ncBackend;
   fSubcontextBackend = cncBackend;
}

// implementation of the inheritance interface
// of NamingContextDefImpl by delegating to fNamespaceBackend

...

// implementation of the inheritance interface
// of CompositeNamingContextDefImpl by delegating to fSubcontextBackend

...

}
```

If a certain combination of inheritance objects is particularly common, that combination can be captured with a separate subclass. The combination of a file-based name space management and a file-based subcontext management would lead to the following definition:

```
public class FileBasedInMemoryCompositeNamingContextFrontImpl
   extends CompositeNamingContextFrontImpl
{

public FileBasedInMemoryCompositeNamingContextFrontImpl()
{
   NamingContextBackend ncBackend =
      new InMemoryNamingContextBackImpl();
   CompositeNamingContextBackend cncBackend =
      new FileBasedNamingContextBackImpl();
   initialize(ncBackend, cncBackend);
}

}
```

A client can then use this well-defined class to create an object directly, without having to write the configuration code.

The principle described here of delegating implementations to separate objects, corresponds to the design pattern Bridge ([2], page 151), for the most part. This design pattern describes how an abstraction (Frontend, for example NamingContext) is parameterized with an implementation (Backend, for example NamingContextBackend). The implementation object could also be considered a Strategy, another design pattern ([2], page 315). However, since strategies are often meant to change during run-time, and this is not an important requirement for this example, Strategy is not used here.

5 Summary

This article shows different aspects of working with Java interfaces and classes, which results from focusing on the modeling of concepts and their implementation. Central topics were clean and robust modeling, as well as achieving optimal code reuse.

The concepts (or patterns) “direct use of classes”, “separation of interfaces and implementation”, “abstract default implementation”, “concrete implementation”, “parallel hierarchies” and “object-based code factoring” allow a fine gradation of flexibility.

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

[3] Sape Mullender. *Distributed Systems.* Addison-Wesley, 1992.

[4] Walter L. Hürsch. “Should Superclasses be Abstract?” In *Proceedings of the 1994 European Conference on Object-Oriented Programming* (ECOOP ’94, LNCS 821). Edited by Mario Tokoro and Remo Pareschi. Springer-Verlag, 1994. Page 12-31.

[5] Dirk Riehle. “Arbeiten mit Java-Schnittstellen und -Klassen”. *Java Spektrum* 6/97 (November/December 1997). Page 35-43.

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.