The Dimension Architecture: A New Approach to Resource Access

Walter Kern, *Landesamt für Finanzen*
Christian Silberbauer, *Competence Center Software Engineering*
Christian Wolff, *University of Regensburg*

A new resource access approach separates various aspects such as address, content format, and location type to enable their flexible and configurable combination.

An important task for almost every software application is I/O. For instance, database applications and even simple applications supporting configuration files—all use I/O. Consequently, accessing and manipulating resources is essential to most software systems. A generic resource access solution can avoid the following problems:

- If resource aspects can't be flexibly combined, the developer must create a new implementation for each combination of resource access aspects, which leads to a combinatory explosion of class implementations. Additional implementations also increase error probability.
- Different application programming interfaces (APIs) for different types of resource locations (such as databases or file systems) or resource formats (such as XML or JavaScript Object Notation[1]) prevent easy substitution of resource access logic used in applications—making modifications time consuming and expensive.
- Inconsistent APIs reduce flexibility by complicating the substitution of implementations at runtime.

We suggest a generic approach based on separating resource access aspects into *dimensions* that can be flexibly combined by configuration. To underline this new concept's necessity, we begin by discussing the current approaches and highlighting the deficits.

Current Resource Access Approaches

Integrating device-specific data access logic in each program was common in the beginnings of third-generation programming languages such as Fortran II. Developers then switched to more flexible concepts when they needed to support multiple data stores concurrently within an application. At the beginning, they used different programming interfaces for different data stores. The variations in implementation weren't restricted to hardware components. Even data stores that varied only in software had to be programmed differently—for example, ADO.NET 1.1, which features a distinct database provider for each database type.[2] In the meantime, plug-in concepts based on uniform programming interfaces such as JDBC were introduced.[3] However, most of these solutions are restricted to spe-

0740-7459/10/$26.00 © 2010 IEEE

cific types of data sources such as databases or file systems.

Plug-in Pattern

The Plug-in Pattern[4] is perhaps the simplest approach to implementing flexible, replaceable resource access. Figure 1 depicts a Plug-in Pattern–based model consisting of a factory class, a configuration resource, and an interface with two example implementations.

The model has a separate class for each type of data source, such as `FileSystemReference` for accessing local files and `JdbcDatabaseReference` for accessing data within a database. All these *references*, or data source providers, implement a common interface. Now, if calling code needs a resource access reference object (such as `JdbcDatabaseReference`), it must call the static `createReference()` method on `ReferenceFactory`, which returns an `IReference` implementation based on the name specified as a parameter of `createReference()`.

This abstraction brings flexibility because calling code doesn't need to know which implementation it operates on. By outsourcing the reference class name to another location, the implementation can be replaced without code changes. The disadvantage is that for each new combination of requirements, the developer must implement a separate reference. If, for instance, a program processes information in three file formats and in four types of data stores, the developer must create 12 implementations for the possible combinations. This causes additional development work, creates a confusing number of classes, and forces the developer to repeatedly reimplement same or similar behaviors.

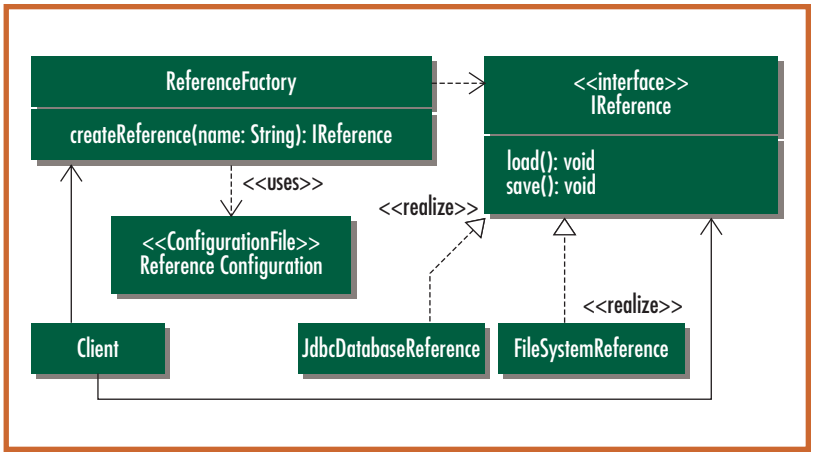

Figure 1. Plug-in Pattern–based resource access model. This model consists of a factory class, a configuration resource, and an interface with two example implementations.

Abstract Factory Pattern

Combining the Plug-in Pattern with the classic gang-of-four style Abstract Factory Pattern[5] (see Figure 2) addresses some of the disadvantages we mentioned earlier. Approaches based on the Abstract Factory Pattern include the Database Factory approach[6] and ADO.NET 2.0.[7]

Basically, concrete factories are derived from an abstract factory, and they return different concrete Reference Instances. Each concrete factory creates a group of similar references relating to a certain aspect—for example, file format. We get the benefit of managing reference groups compared to the nonclassified plug-in concept approach. Nonetheless, the main issues remain—we must write a distinct class for each requirement combination because we can't combine the various requirement types (such as XML as resource format and network as resource location type). Moreover, the Abstract Factory Pattern approach only lets us discriminate by one criterion at a time—the family criterion (for example, file format).

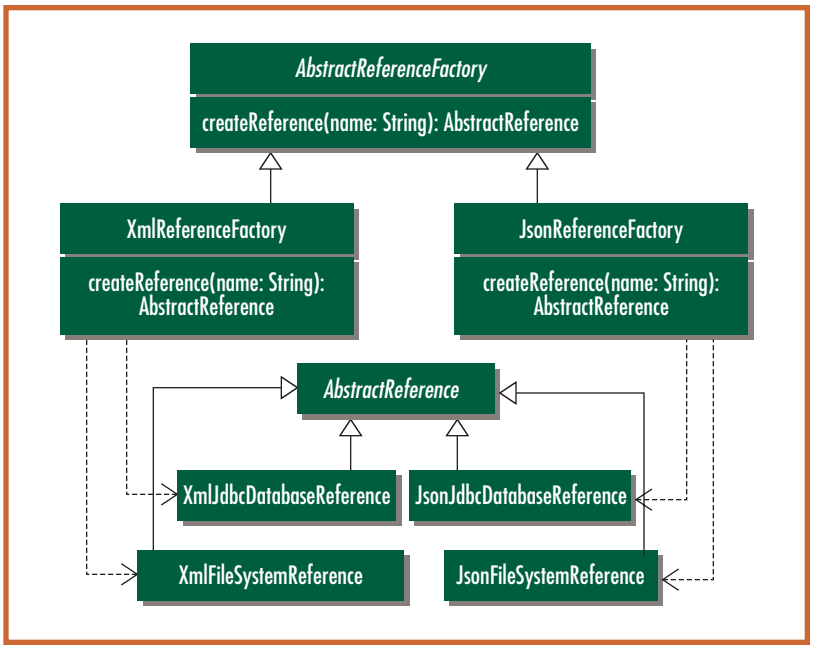

Figure 2. Abstract Factory Pattern–based resource access approaches consist of an abstract factory, derived concrete factories, an abstract resource access reference, and concrete reference implementations.

Decorator Pattern

The Decorator Pattern,[5] another classic gang-of-four pattern, lets us dynamically add functionality to an existing object using *decorator classes* that inherit from the same base class and can be nested to enhance their functionality. This approach eliminates the combinatory explosion of classes when combining distinct behaviors.

In terms of resource acquisition, the Java I/O architecture[8] is a well-known application of this

No current approach can fully satisfy the requirements for generic and flexible resource access.

pattern. The corresponding nested decorators, also called *stream filters*, build the I/O stream. An example of InputStream-based nesting is

```
BufferedInputStream b=new BufferedInputStream(
    new FileInputStream(new File("path")));
```

This resolves the combinatory explosion problem but introduces new issues because it lets us combine incompatible and inconsistent behaviors (for example, we could join a hypothetical XMLReader with a BinaryReader). Furthermore, some decorators might need a specific nesting order—for example, a stream class handling the actual resource access might need to execute first. However, the current concept doesn't support explicitly specifying the order. We must use documentation to provide this information, which rules out tool-based validation. Finally, the Decorator Pattern doesn't have a built-in classification by distinct aspects such as file format or location type.

Resource Acquisition Approaches

The approaches we've mentioned so far are essentially general-purpose patterns. Additionally, a few important patterns directly focus on resource acquisition and access.

The URL-based approach. URLs[9] contain a protocol identifier that describes the location type and protocol-specific address. For example, the URL "http://www.computer.org/software" includes the protocol "http" and references a resource with the name "www.computer.org/software." In Java, we create URLs by using the java.net.URL class. This class uses a java.net.URLStreamHandlerFactory to get implementations of a java.net.URLStreamHandler, which contains address-parsing logic and returns an instance of a java.net.URLConnection implementation that contains location access logic. Consequently, we can implement arbitrary address formats and location types.

We can combine this approach with Multipurpose Mail Extensions.[10,11] In Java, the factory java.net.ContentHandlerFactory returns java.net.ContentHandler instances, which represent different MIME types. For this reason, this approach also supports arbitrary resource content types in the form of different MIME types, and resource access and content handling are decoupled and orthogonal.

However, the separation between address and location aspects is shallow because the location access implementation (a java.net.URLConnection) is created from within the address logic implementation (a java.net.URLStreamHandler) that's mapped to a URL's protocol identifier.

So, owing to this hardwiring, we must define a new protocol identifier and URLStreamHandler for each new combination of an address format and a location type. This creates code redundancy—if a specific location type supports n address formats, we must create n URLStreamHandler implementations with the same address-parsing logic but different URLConnection implementations. Considering m location types that support n address formats, $n \times m$ implementations must be created. This leads to a combinatory explosion problem as well.

Regarding further resource access aspects such as security and logging, we can manipulate URLs using Java stream filters. On one hand, this Decorator-based approach avoids the combinatory explosion issue in the context of extended resource access aspects. But on the other, it introduces the disadvantages of Decorator-based approaches that we mentioned earlier.

Other approaches. In the Java Naming and Directory Interface (JNDI),[13] the Service Locator Pattern[12] describes distributed service object lookups and provides a central point of control. Even if the lookup process is more flexible compared to the Decorator Pattern, it shares the other patterns' problems, including the combinatory explosion problem. Additionally, it doesn't deal with resource load or save mechanisms. It only describes a way to get service and resource references. In comparison to the Service Locator Pattern, the Lookup Pattern[14] isn't tailored for a certain technology such as JNDI. Nonetheless, it shares nearly all the disadvantages of the Service Locator Pattern (such as the combinatory explosion problem and the missing separation of resource access aspects).

So, no current approach can fully satisfy the requirements for generic and flexible resource access. But what if we could find a generic solution that lets us combine not only the requirement types of zero (Plug-in Pattern), one (Abstract Factory Pattern), or three dimensions (URL based approach), but any number of dimensions—and, unlike the Decorator Pattern, reduces the combination of noncombinable resource aspects? Our Dimension Architecture aims to overcome current solutions' limitations.

The Dimension Architecture

Dimensions are the core aspect of the Dimension Architecture approach. By *dimension*, we can mean either *dimension types* or *dimension instances*, depending on context.

Dimension instances are concrete resource access aspects. For example, a *dimension instance* can interpret XML or read a file-system resource.

They're configurable by attribute-value pairs specific to the purpose of the respective dimension instance—for example, an XML dimension instance would likely support an *Encoding* parameter.

Dimension types categorize dimension instances by their domain. Three basic dimension types are essential to resource access:

- *Address* describes dimension instances that address resources in a specific address format, such as Address Dimension Instances for file-system resources in Windows file path format (C:\programs) and UNC format (\\?\C:\programs\).
- *Location* type describes the location type, such as file system, database, or network.
- *Format* describes dimension instances that deal with resource formats, such as Format Dimension Instances for XML or binary format.

Unlike with basic dimension types, *extended dimension types* aren't predefined, because they're not essential for all resource access scenarios. Different resource access scenarios could require different extended dimension types and extended dimension instances because there might be additional resource access requirements, such as authentication protocol support.

As Figure 3 shows, our approach combines exactly one dimension instance of each basic dimension type and an arbitrary number of extended dimension instances as a *Reference Instance*, which is used for the actual resource access. A Reference Instance contains an *Internals structure* that comprises the properties `Address`, `Content`, and `Document`, which represent characteristics of the referenced resource during different processing stages (see the "Resource Access" section below).

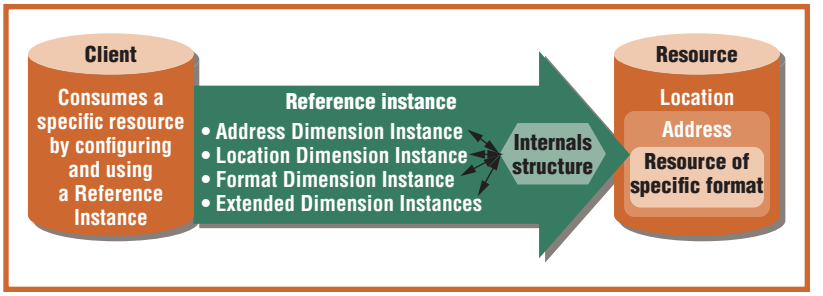

Figure 3. The Reference Instance and its components. The Reference Instance and its configured dimension instances are used to access a specific resource.

Regarding Reference Instance management, a reference factory returns a default Reference Instance implementation with the specified ID and of the configured type on the basis of a corresponding definition in the specified configuration resource (the default is the application's local configuration file). Furthermore, our approach supports custom implementations of Reference Instances to add additional methods that operate on the resource's object-oriented representation and allow working with the resource as a business entity.

Resource Access

Data are loaded and saved by calling `load()` and `save()` of the corresponding Reference Instance. Consequently, the corresponding dimension instances' `process()` methods are sequentially called, and the dimension instances are processed. Figure 4 shows the process resulting from the call to a Reference Instance's `load()` method.

First, the Reference Instance's Address Dimension Instance transforms the configured address to a representation that the configured Location Dimension Instance understands. The `Address`

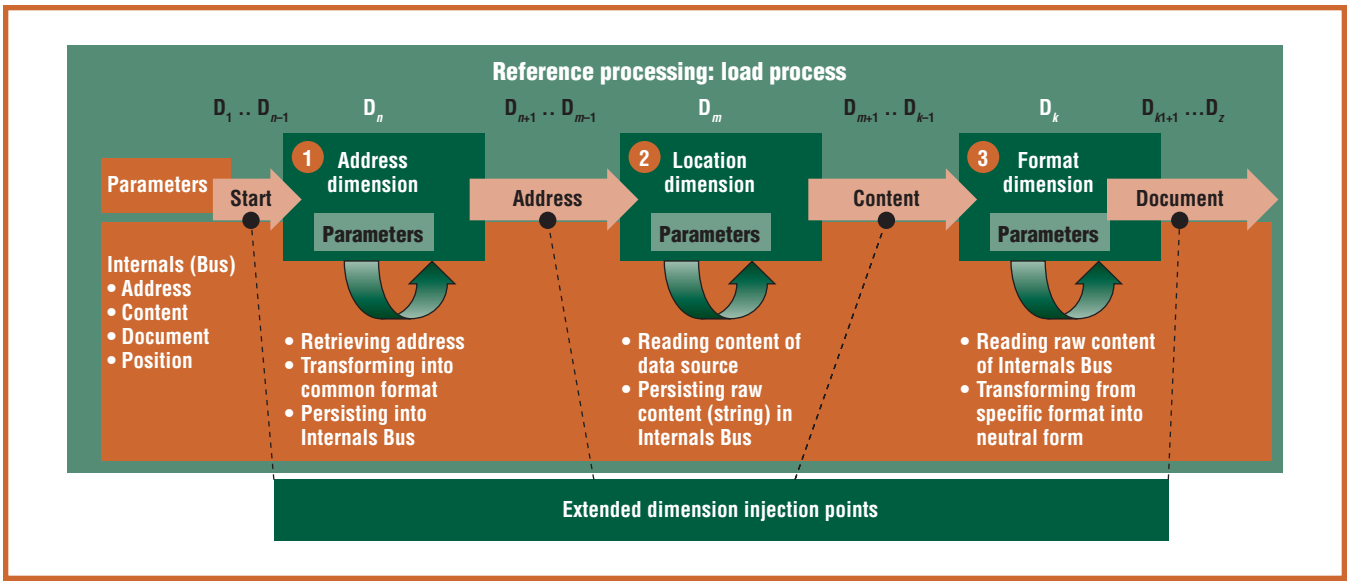

Figure 4. Processing pipeline of reference load process. The reference load process pipeline is used to load data from a specific resource.

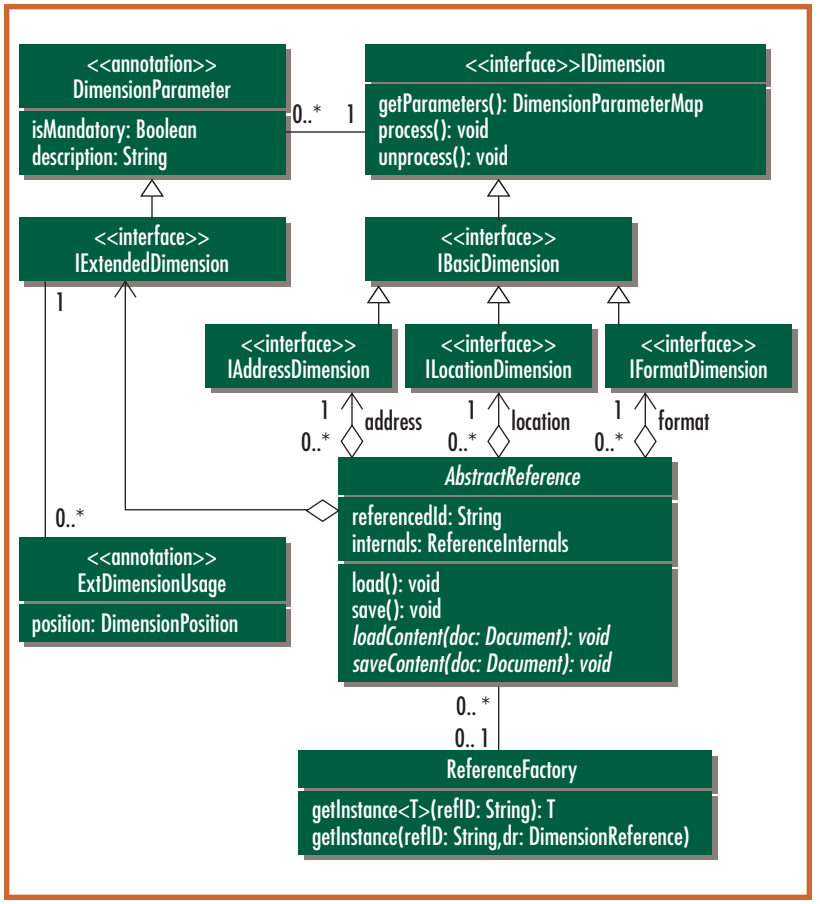

Figure 5. Simplified class diagram of major Dimension Architecture entities and the relationships among them.

property of the Internals structure stores the calculated address. After the transformation and storing of the calculated address, the Reference Instance's Location Dimension Instance executes and loads the referenced resource's raw content into the Internals structure's `Content` property. Finally, the configured Format Dimension Instance executes and takes the content saved in the `Content` property, transforms it in a structured hierarchical form, and saves the result in the Internals structure's `Document` property.

Additionally, we can process extended dimension instances between each of the steps we've mentioned. For example, if the raw content is encrypted, we might implement an extended dimension instance with encryption support. We could configure this implementation to be called between the processing of the Reference Instance's Location Dimension Instance and the Format Dimension Instance.

Basically, the Reference Instance's save process is implemented as the reversal of the load process. This means that changes made to the Internals structure's `Document` property are written back to the resource by calling the dimension instance's `unprocess()` functions in the reversed execution order.

Execution Order of Dimension Instances

Dimension instances execute in the order in which they're defined in the corresponding configuration resource. However, a Reference Instance's Address Dimension Instance must be processed before a Location Dimension Instance, which itself must execute before a Format Dimension Instance. This is ensured by a runtime check that's integrated in the generic implementation of the Reference Instance and terminates the execution if the required order is violated.

This check also ensures the correct configuration order of extended dimension instances. The developer specifies the valid position using class annotations that indicate valid positions relative to the positions of the Reference Instance's chosen basic dimension instances. The following positions (extended dimension injection points) are supported (see Figure 4):

- After Start: execution before the Address Dimension Instance.
- After Address Dimension Instance: execution after the Address Dimension Instance and before the Location Dimension Instance.
- After Location (Type) Dimension Instance: execution between the Location Type Dimension Instance and the Format Dimension Instance.
- After Format Dimension Instance: execution after the Format Dimension Instance.

Now that we have explained the concepts behind the Dimension Architecture approach, we will illustrate the concepts by showing our approach's reference implementation.

Reference Implementation

Figure 5 illustrates the Dimension Architecture's core elements. Table 1 shows the mapping of main conceptual terms to the corresponding implementation.

Example and Code Sample

In this section, we illustrate the Dimension Architecture with a Java code example based on our Reference Implementation. The task is to load customer data from an XML file on a Microsoft Windows–based network. The content is encrypted with a dummy algorithm and shown below (in clear text).

```
<?xml version="1.0" encoding="UTF-8"?>
<Customers>
  <Customer>
    <LastName>Doe</LastName>
    <FirstName>John</FirstName>
```

```
    <Gender>m</Gender>
  </Customer>
  <Customer>
    <LastName>Mustermann</LastName>
    <FirstName>Max</FirstName>
    <Gender>m</Gender>
  </Customer>
</Customers>
```

This sample XML file contains customer data that is going to be processed.

Configuration. To process the sample file with the customer data, we create a configuration file and define a Reference Instance with the id *r1*. Then we define the resource access aspects and configure an `UncAddressDimension` with the address \\?\C:\Customers.xml. Next, we add a `FileSystemLocationDimension` because our resource is on the file system; we also use an `XmlFormatDimension`. Finally, we configure extended dimension instances as specified by additional requirements: a `DummyEncryptionExtendedDimension` and a symmetric key as parameter. Because the resource content is encrypted and the `XmlFormatDimension` expects unencrypted XML data, we must position this extended dimension instance before the Format Dimension Instance.

Here, we show the final configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<References>
  <Reference id="r1" type="demo.CustomerReference">
    <AddressDimension type="dimension.impl.
                    UNCAddressDimension">
      <Param name="Address" value=
                    "\\?\C:\Customers.xml"/>
    </AddressDimension>
    <LocationDimension type="dimension.impl.
                    FileSystemLocationDimension"/>
    <ExtendedDimension type="dimension.impl.
                    DummyEncryptionExtendedDimension">
      <Param name="SymmetricKey" value="xyz"/>
    </ExtendedDimension>
    <FormatDimension type="dimension.impl.
                    XmlFormatDimension"/>
  </Reference>
</References>
```

This configuration file is used to specify all required dimensions to reference the customer file we mentioned earlier.

Custom Reference Instance Implementation. To work with a resource in a business-entity style, we implement an entity class `Customer` and a *Reference Instance* that inherits from `AbstractReference` and overrides several of its methods, returning customer entities:

Table 1

Mapping between Dimension Architecture terms and reference implementation entities

Dimension Architecture term	Reference implementation entity
Reference instance	`AbstractReference`
Reference factory	`ReferenceFactory`
Basic dimension instance	`IBasicDimension` implementation
Extended dimension instance	`IExtendedDimension` implementation
Address Dimension Instance	`IAddressDimension` implementation
Location Dimension Instance	`ILocationDimension` implementation
Format Dimension Instance	`IFormatDimension` implementation

```
public class Customer{
  public String FirstName,LastName, boolean MaleGender;
  //…Constructors
  @Override public String toString() {
    return String.format("%s %s (%s)",FirstName,
    LastName,MaleGender?"male":"female");
  }
}

public class CustomerReference extends AbstractReference{
  private ArrayList<Customer>
                    customers=new ArrayList<Customer>();
  public ArrayList<Customer> getCustomers() {return customers;}

  @Override protected void loadContent(Document doc)
                    throws Exception {
    //Parse Document property of Internals Structure +
                    build business entities
    for (int k=0;k<doc.getChildNodes().getLength();k++) {
    customers.add(new Customer(…));
                    //new Customer based on Node
    }
  }

  @Override protected void saveContent(Document doc)
                    throws Exception {
    //Save business entities in Document property
                    of Internals Structure
    Node rootNode=doc.appendChild(doc.
                    createElement("Customers"));
    for (Customer c:customers) {
```

The Dimension Architecture supports an arbitrary number of dimensions for resource access.

```
        Node customerNode=doc.createElement("Customer");
        …
        rootNode.appendChild(customerNode);
                        //new Node based on Customer
      }
    }
}
```

This code illustrates the implementation of a business-logic-specific Reference Instance.

Client Implementation. Using Reference Instances requires a few steps: the reference factory builds a Reference Instance and returns it on the basis of the configuration file and the specified ID. After that, we can manipulate the resource by modifying the corresponding business entities. A call to save() writes modifications of the business entities back to the referenced resource:

```
CustomerReference reference=ReferenceFactory.getInstance("r1");
reference.load();//Load content

//Iterate over customers
for (Customer c:reference.getCustomers())
   System.out.println(c);

//Manipulate customers(in memory)
reference.getCustomers().add(new Customer("Julian","Gerak",true));

reference.save();//Save customers back to resource
```

This code shows the usage of the Dimension Architecture for loading, manipulating, and saving a resource based on business entities.

Process Description. To access a resource based on the Reference Instance configuration we explained earlier, we call getInstance() of the reference factory and specify the reference identifier. A call to the reference's load() method executes the process() methods of all configured dimension instances.

The process() method of UncAddressDimension is called, and it translates the specified address (\\?\C:\Customers.xml) to a common address format and saves it in the Internals structure's Address property. Next, the configured FileSystemLocationDimension reads the raw content from the calculated address in the internal structure's Content property. After that, the DummyEncryptionExtendedDimension replaces the Content property value with its decrypted equivalent. Afterward, the XmlFormatDimension reads the unencrypted content and saves a hierarchical representation of it in the internals structure's Document property.

Thereafter, the Reference Instance's loadContent() function is called and creates corresponding business entities that the client can manipulate. A call to save() saves the modifications back to the resource by calling unprocess() of all configured dimension instances in the reverse order.

Dimension Implementation. Basically, all that must be done to implement resource access is to configure a Reference Instance, derive from the AbstractReference, and provide the client implementation. If a certain aspect isn't available as a dimension instance, it can be implemented. The following code shows the skeleton of a DummyEncryptionExtendedDimension. As we mentioned earlier, implementations of IExtendedDimension have two methods—process() and unprocess(). Process() is called on load processes of the Reference Instance and unprocess() is called on save processes.

```
@DimensionParameter(name="symmetricKey",mandatory=true)
@ExtDimensionUsage({DimensionPosition.AfterLocation})
   public class DummyEncryptionExtendedDimension implements
   IExtendedDimension {
   @Override public void process(){
      String symmetricKey=parameters.get("SymmetricKey");
      //decrypt internals.Content+write decrypted text back
                        to internals.Content
   }
   @Override public void unprocess(){
      String symmetricKey=parameters.get("SymmetricKey");
      //encrypt internals.Content+write encrypted text back
                        to internals.Content
   }
}
```

This code shows the implementation of a custom extended dimension.

In this sample code, we annotated the extended dimension with valid execution positions relative to the basic dimensions. In this case, the custom dimension instance can be executed after the Location Dimension Instance (and, as a consequence, before the Format Dimension Instance) in the load process. For the save process, the execution order of all configured dimension instances is reversed.

Our Dimension Architecture approach's main advantage is its flexibility, because any resource access aspect can be substituted by configuration or at runtime. Compared to Decorator Pattern–based approaches, incompatible and unreasonable combinations of resource-access-related aspects can be restricted

because a Reference Instance always contains exactly one instance of each Basic Dimension Type. Furthermore, the order of resource aspect execution is configurable and verified by a runtime check. In contrast to the URL approach, additional resource access requirements can be added by configuring and optionally implementing additional (Extended) Dimension Instances. Unlike most other approaches, the Dimension Architecture also avoids combinatory explosion. Another advantage is that our approach is easy to comprehend and learn because for typical usage scenarios it is sufficient to configure existing Address, Location Type, and Format Dimension Instances and to instantiate a Reference Instance in the client code. Required Dimension Instances can be selected using the mnemonic device "ALF"—Address, Location Type, Format. Finally, the declarative and annotation-based nature of the Dimension Architecture enables the creation of tools for users and developers to build Reference Instances graphically.

Our Dimension Architecture approach also has some disadvantages. The application probably results in slightly slower runtime performance compared to raw, fixed-resource access such as FileReader for local files in Java. In most cases, though, the additional flexibility is more relevant than small performance loses.

Another drawback is the lack of resource content streaming because the approaches' multistage process requires complete intermediate results to work on. This shortcoming could be softened by replacing the Internals structure's elements with proxy classes that implement lazy and partial loading.

The Dimension Architecture is highly flexible. Even if it seems to have the same potential for abstraction as the Resource Acquisition Pattern,[14] it's too early to make a classification claim yet. To verify its potential, we will use it in larger projects, including BeihilfeOnline, a Web 2.0-based e-government application for claiming refunds for medical treatment expenses targeting about 300,000 users in the Bavarian public service sector. Additionally, we plan to submit our Java implementation of the Dimension Architecture to the Java Community Process.

References

1. D. Crockford, *The Application/JSON Media Type for JavaScript Object Notation (JSON)*, IETF RFC 4627, July 2006; www.rfc-archive.org/getrfc.php?rfc=4627.
2. Microsoft Corp., ".NET Framework Developer's Guide: .NET Framework Data Providers," 2006; http://msdn.microsoft.com/en-us/library/a6cd7c08(VS.71).aspx.
3. L. Andersen, *JSR-000221 JDBC 4.0*, v. 2.6, Java Community Process specification, Nov. 2006; http://jcp.org/aboutJava/communityprocess/final/jsr221/index.html.
4. M. Fowler, D. Rice, and M. Foemmel, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002.
5. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
6. A.R. Selvaraj and D. Ghosh, "Implementation of a Database Factory," *ACM SIGPLAN Notices*, vol. 32, no. 6, 1997, pp. 14–18.
7. D. Sceppa, *Programming Microsoft ADO.NET 2.0 Core Reference*, Microsoft Press, 2006.
8. E.R Harold, *Java I/O*, O'Reilly & Associates, 1999.
9. T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, STD 66, RFC 3986, Jan. 2005; www.rfc-archive.org/getrfc.php?rfc=3986.
10. N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, IETF RFC 2045, Nov. 1996; http://rfc-ref.org/RFC-TEXTS/2045/index.html.
11. N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*," IETF RFC 2046, Nov. 1996; http://rfc-ref.org/RFC-TEXTS/2046/index.html.
12. Sun Microsystems, *Java Naming and Directory Interface Application Programming Interface*, Nov. 1999; http://java.sun.com/j2se/1.5/pdf/jndi.pdf.
13. W. Crawford and J. Kaplan, *J2EE Design Patterns*, O'Reilly Media, 2003.
14. M. Kircher and P. Jain, *Pattern-Oriented Software Architecture: Patterns for Resource Management*, John Wiley & Sons, 2004.

About the Authors

Walter Kern is a software architect at the Landesamt für Finanzen in Regensburg and a PhD student in information science at the University of Regensburg. His research interests include software engineering, distributed systems, and Web technologies. Kern has a Diploma in computer science from the University of Applied Sciences in Regensburg. Contact him at walter.kern@lff.bayern.de.

Christian Silberbauer is a software architect at the Software Engineering Competence Center and a PhD student in information science at the University of Regensburg. His research interests include the design of programming languages. Silberbauer has a Diploma in computer science from the University of Applied Sciences in Regensburg. Contact him at christian.silberbauer@cc-se.de.

Christian Wolff is an associate professor of media computing at the University of Regensburg. His research interests include software engineering, interface design, Web-based systems, and information retrieval. Wolff has a PhD in information science from the University of Regensburg and a habilitation degree in computer science from the University of Leipzig. He's a member of the IEEE Computer Society and the ACM. Contact him at christian.wolff@computer.org.