Rearchitecturing Legacy Systems—Concepts & Case Study

Wolfgang Pree
Software Engineering Group
University of Constance
D-78457 Constance, Germany
pree@acm.org
www.altissimo.com

Kai Koskimies
Nokia Research Center
Box 45
FIN-00211 Helsinki, Finland
kai.koskimies@research.nokia.com

Abstract. Legacy systems, no matter which architectural style they rely on, contain numerous pieces of source code with very similar functionality. We see these system aspects as a good starting point for rearchitecturing legacy systems. The goal is the evolution of the legacy system architecture towards a product line architecture which incorporates the originally replicated system aspects as reusable, ideally self-configuring components. This paper presents the concepts which we regard as necessary and/or useful for such an evolution: Framelets form small architectural building blocks that can be easily understood, modified and combined. Reflection together with a high-level definition of semantic aspects allow the construction of partially self-configuring components. A case study corroborates that this constitutes a viable approach for rearchitecturing legacy systems in practice.

Keywords: creation and evolution of architectures, product line architectures, framelets, automated configuration, dynamic architectures.

1 Product line architectures for replicated components

The source code of legacy systems comprises numerous replications of similar chunks of code. This means that from an architectural perspective many components of the overall architecture provide similar if not identical functionality. In other words, source code was written again and again from scratch for implementing these components. The idea for rearchitecturing legacy systems suffering from this problem is to develop a product line architecture for each such replicated component. The particular components are slight variations of the product line, that is, they belong to the family of the product line. Depending on the size of the replicated components, this kind of rearchitecturing activity will lead to a set of small product lines.

Let us take a look at a specific legacy system which we rearchitectured recently[1]. The three-tier client/server (CS)–system of the bank is representative of legacy software systems relying on the CS-architectural style. The clients (Windows PCs) access a central data repository via a remote procedure library, which is available as set of C functions. The data repository resides on a server machine (currently a Unix workstation; migration to Windows NT is under way) and/or mainframe. The remote procedure library represents a quite stable part of the system architecture which has not changed at all over the past ten years. For implementing the client side the bank used a Fourth Generation Tool (SQL Windows/Gupta). The problem associated with this approach is that the tool produces a monolithic architecture: all dialog windows form one executable which has to be loaded to each client no matter how small the percentage of required dialogs actually is. From a development point of view it is hardly possible to package dialogs or parts of dialogs into reusable components.

From an architectural point of view the module structure of the client system is quite natural to envision. One dialog forms one module. In some cases a small group of dialogs might be packaged into one unit. Despite the shortcomings of typical Fourth Generation Tools regarding modularization, several other choices, such as state-of-the-art Java development environments allow the straightforward implementation of such a module structure.

A closer look at the module structure of dialogs reveals that almost every such module contains one or more components for handling remote procedure calls and one or more components for managing items in a list. Of course, the components differ in various contexts. For example, before a remote procedure is invoked, the input parameters of the procedure have to be read out of specific GUI elements. The number of parameters and the GUI elements differ between remote procedure calls (RPCs). RPCs return their results in C-arrays that have to be interpreted properly. The results are then displayed again in GUI elements. This infrastructure surrounding an RPC is an example of source code that has to be implemented again and again for each RPC, but that offers similar functionality.

As most dialogs in real world CS-systems have one or more GUI elements that display items in lists (by means of a GUI component called multi-column grid control), interactions associated with lists are also replicated in most dialogs. For example, a button for removing an item a the list has to be enabled only if an item in the list is selected, otherwise the button is disabled. Pressing a button to add an item opens a dialog window for entering the data. Pressing a button to modify an item also opens a dialog window and transfers the data representing the item to the corresponding GUI elements for the purpose of editing them. The aspects that differ in the various list handling components are the types of the listed items, the dialog window to display an item and some details such as button labels and the location of buttons for manipulating the list (for example, under the grid control or beside of it).

Figure 1 illustrates schematically the problem of replicated components in the architecture of the CS-system at hand. Though the size of these components is small (about 200 to 300 source lines of code), they are replicated several hundred or even thousand times. Note that Figure 1 shows

---

[1] The project is part of a cooperation between RACON Software GmbH, a software house of the Austrian Raiffeisen bank, and the Software Engineering Group at the University of Constance. The principal question at the outset was, wether a rearchitecturing effort based on framework technology and Java can lead to a significantly better modularization of the overall system that allows the reuse of components. The project was a clear success. The paper presents those concepts and ideas which we regard as generally useful for rearchitecturing legacy systems.

several replicated RPC components, but only one ListHandling component, as just one of the two sample dialogs contains a list. (Figures 1 and 2 apply the notation introduced by Bass et al. (1998). Solid arrows express control flow, dotted arrows depict data flow.)

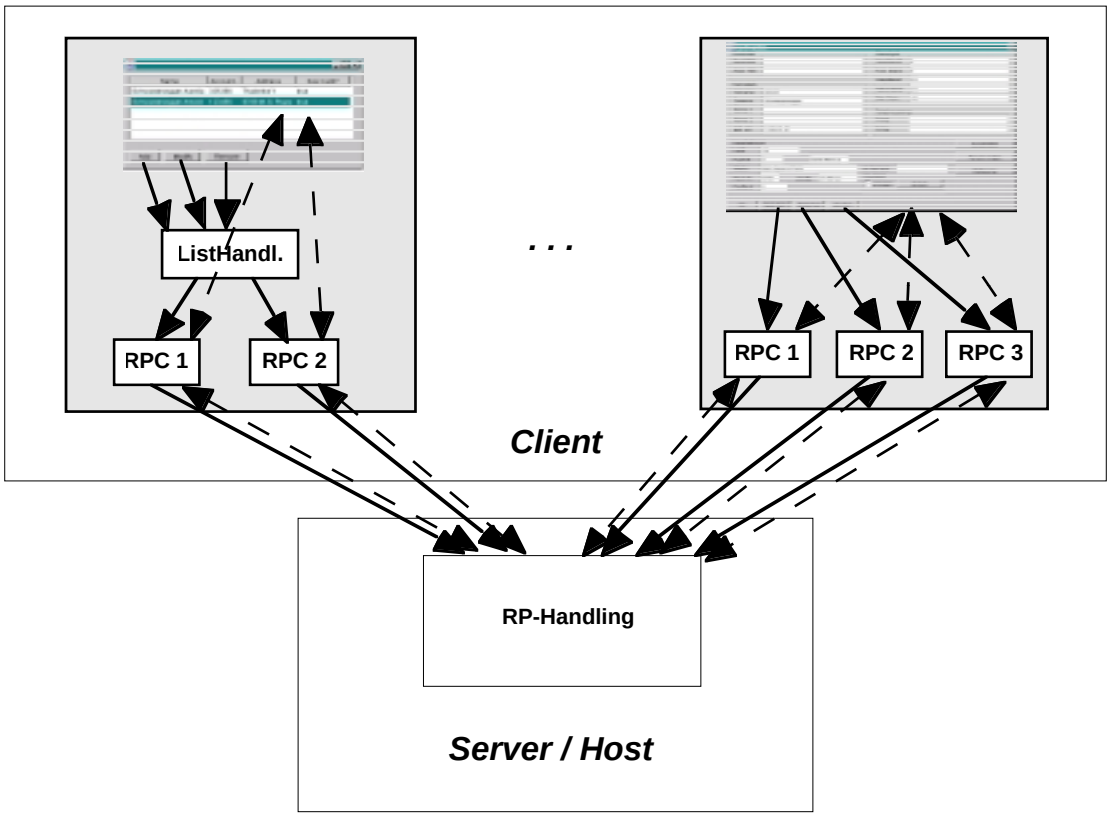

Figure 1 Module structure of the CS-system with replicated RPC components.

Figure 2 shows an architectural solution which is based on a small product line for each such component. This solution is better as the number of components is significantly reduced.

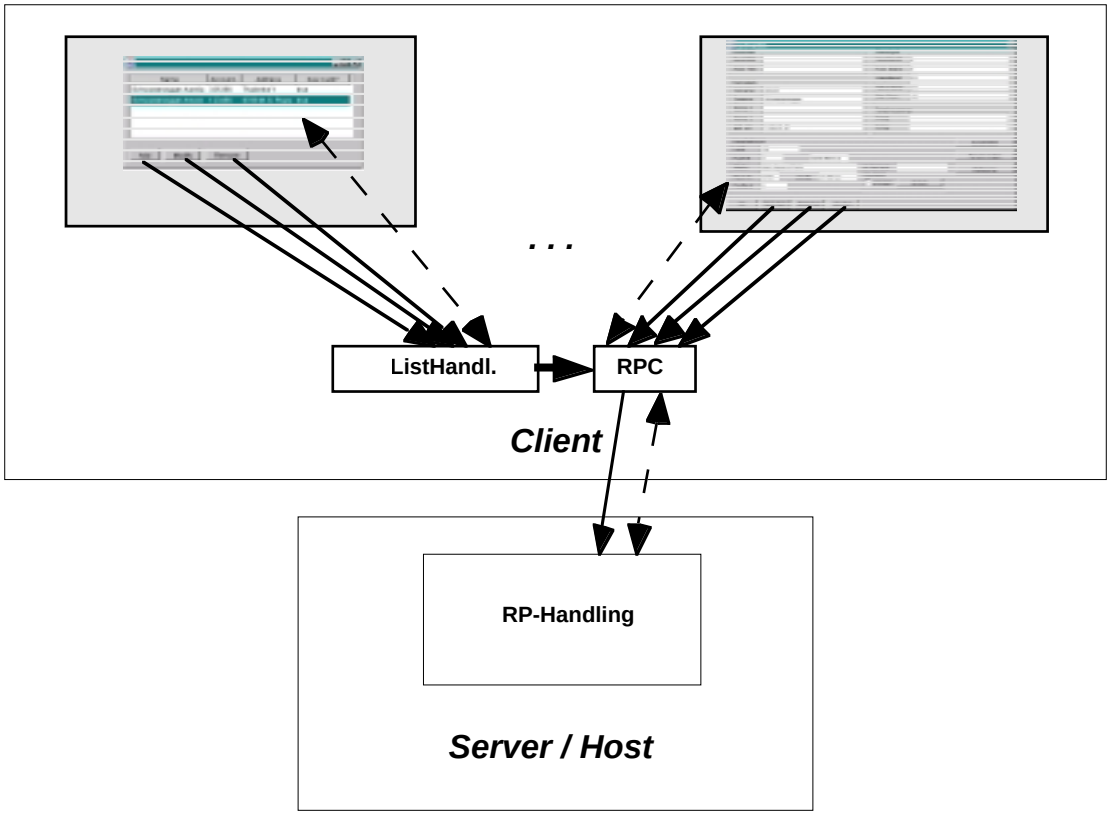

Figure 2 Module structure of the CS-system with a product line architecture.

The following sections present the concepts underlying reflection-based framelets. Such framelets were used to develop the product line architecture sketched above. A case study discussing the RPC-framelet concludes the paper.

2 Framelets

Object-oriented frameworks can be of any size, ranging from even one or a few simple classes to large sets of complex classes. However, the conventional idea of a framework is that it constitutes the skeleton of a complex, full-fledged application. Consequently, frameworks tend to be relatively large, consisting of, say, hundreds or thousands of classes. We argue that the common problems (see e.g. Casais (1995); Sparks et al. (1996); Bosch et al. (1998)) associated with frameworks stem from this idea:

We argue that the reason for common problems associated with frameworks is the conventional idea of a framework as the skeleton of a complex, full-fledged application:

- The design of such typical frameworks is hard. Due to the complexity and size of application frameworks and the lack of understanding of the framework design process, frameworks are usually designed iteratively, requiring substantial restructuring of numerous classes and long development cycles.

- Reuse of a framework is hard. A framework conventionally consists of the core classes of an application, and one has to understand the basic architecture of a particular application type to be able to specialize the framework.
- The combination of frameworks is hard. Often a framework assumes that it has the main control of an application. Two or more frameworks making this assumption are difficult to combine without breaking their integrity.

A framework becomes a large and tightly interconnected collection of classes that breaks sound modularization principles and is difficult to combine with other similar frameworks. Inheritance interfaces and various hidden logical dependencies cannot be managed by application programmers. A solution proposed by many authors is to move to black-box frameworks which are specialized by composition rather than by inheritance. Although this makes the framework easier to use, it restricts its adaptability. Furthermore, problems related to the design and combination of frameworks remain.

This suggests that not the construction principles of frameworks form a problem, but the granularity of systems where they are applied. We propose a radical downsizing of frameworks and call these assets *framelets*. In contrast to a conventional framework, a framelet

- is small in size (< 10 classes),
- does not assume main control of an application, and
- has a clearly defined simple interface.

Like conventional frameworks, a framelet can be specialized by subclassing and composition.

We consider a framelet not only as a reusable asset but indeed as a fundamental unit of software in general. If a software system is seen as a set of service interfaces and their implementations, a framelet is any (small) subset of such a system. An interface that belongs to the framelet without its implementation (and used within the framelet) is part of the specialization interface of the framelet. An interface that belongs to the framelet together with its implementation (and used ouside of the framelet) is part of the service interface of the framelet. This is basically the foundation for using framelets in restructuring legacy systems.

Our vision is to have a family of related framelets for a domain area representing an alternative to a complex framework. Thus we view framelets as a kind of modularization means of frameworks. On a large scale, an application is constructed using framelets as black-box components, on a small scale each framelet is a tiny white-box framework.

A particular problem arising from the use of framelets as production lines is *specialization dependency*: the problem of specializing a large conventional framework may reappear in the case of framelets as the existence of various hidden dependencies that the specializations of individual framelets must follow to build a consistent application. Ideally, it should be possible to specialize each framelet independently of the others. To make this possible, framelets should be able to adapt themselves automatically to the context in which they are being used, relieving the programmer of the burden of explicitly writing the context requirements as configuration code in the specialization. In the sequel we show that this can be at least partially achieved using reflective features provided by many OO languages (e.g. Java), together with certain semantic conventions.

3 Reflection as basis of self-configuring assets

What frameworks and framelets have in common is that they represent one means of implementing product line architectures. For this purpose they rely on the constructs provided by object-oriented programming languages. The few essential framework construction principles, as described, for example, by Pree (1996), are applicable to framelets as well. A framelet retains the call-back characteristic (aka Hollywood principle) of white-box frameworks: framelets are assumed to be extended with application-specific code called by the framelet. Figure 3 (a) shows a run-time snap shot of a framelet with the objects A and B as hot spots. Usually hot spots correspond to abstract classes or Java interfaces in the static program code. A reuser of the framelet would have to choose either from already existing specific subclasses of the abstract classes or from interface implementations, or would have to implement appropriate classes. The framelet is adapted by replacing the place holders by instances of specific A and B classes (see Figure 3 (b)).

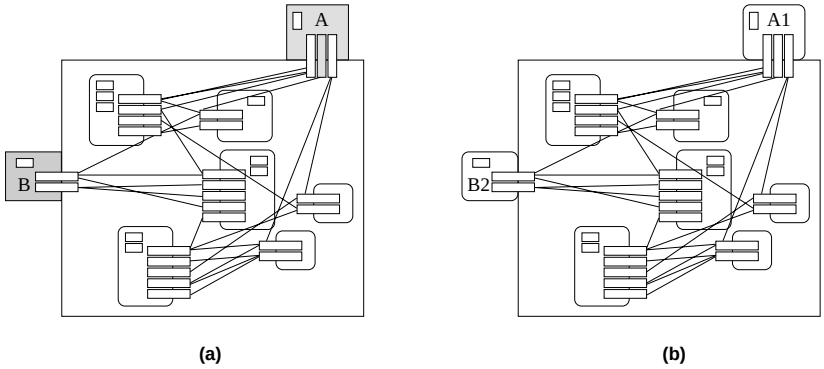

Figure 3 Framelet before (a) and after (b) adaptation.

Besides the mentioned canonical possibilities of defining abstract entities of a framelet, there exist significantly more flexible ways of doing this, albeit they sacrifice type safety. Let us assume we design the framelet sketched in Figure 3 in Java, where all classes have a common ancestor, i.e., they inherit from class Object. Now the framelet designer could decide not to restrict the two hot spots to a specific type, such as A and B in our example. Instead it should be possible to plug any object into the framelet. In other words, the static type of these hot spots becomes Object. The only useful assumption that the framelet designer can make about these abstract entities is that they provide the full range of meta-information. As meta-information is supported by the Java standard library (JDK 1.1 and above) any object offers the same range of meta-informations. For example, it becomes possible to iterate over instance variables, access their types and values, iterate over an object's methods and invoke particular ones. On first consideration, this seems to be useless as no semantics are associated with these operations, as opposed to abstract classes or interfaces, whose methods define a specific type with an associated behavior on which the framework developer can rely.

The advantage of such reflection-based hot spots is that a little bit 'intelligent' framelets can be constructed that exhibit self-configuring properties. The framelet generically couples itself with the objects that fill the hot spots. In order to make this happen, some semantics have to be defined for the abstract entities. The sample framelet discussed in the next section applies a very

simple mechanism for defining semantics, i.e., naming conventions. The point is that the semantic definitions are completely decoupled from the programming language level. They reintroduce a notion of typing on a more domain-related level. Thus proper semantic definitions render void the above mentioned drawback of giving up strong typing. They introduce kinds of equivalents of types on the domain level. Of course, naming conventions are probably the most basic means of defining semantics. We are currently investigating more sophisticated means of pragmatically defining domain-specific semantics.

4 The RPC product line—a case study

Remember that calling a remote procedure requires some infrastructure in addition to the mere invocation. The values of the input parameters of the remote procedure originate from GUI elements. The return parameters of most remote procedures are packaged in a C-array that has to be carefully processed before they can be displayed in particular GUI elements.

The interface of a reusable asset should be designed as straight-forward for the user as possible. If the infrastructure surrounding an RPC is packaged in a reusable asset, the ideal situation would be that the reuser just invokes one method, doRPC(...), of this component. The first parameter is the name of the RPC as a string. The second parameter of doRPC(...) is a reference to the dialog window which contains the GUI elements corresponding to the input parameters of the remote procedure. Finally, a reference to the dialog window has to be specified in whose GUI elements the result parameter values are displayed. Let's call these two dialogs input and output dialog windows. Note that the input and output dialog windows can be identical. The RPC component should ideally be able to do the configuration job itself, i.e., extract the parameter values from the appropriate GUI elements of the input dialog window and transfer the results to the GUI elements of the output dialog window. This would make the reusble asset a perfect small product line for calling remote procedures. How can such a convenient reuse level be achieved?

Here a simple naming convention comes into play. The GUI elements have to have the same names as the RPC parameters. The RPC component is implemented as a framelet in Java with two core hot spots: the input dialog window and the output dialog window. Both hot spots are of type Object. As discussed in detail below, the RPC framelet only requires the meta-information interface to accomplish the configuration job. We'll see that the naming convention is a sufficient semantic specification of the behavior of the two hot spots.

The RPC product line works internally as follows: The framelet is based on a parameter description for each remote procedure. The type of each parameter of a particular remote procedure has to be known. Furthermore, a parameter has to be classified as an input or an output parameter. In the realm of the RPC framelet, the class construct was chosen to describe a remote procedure. (These classes don't have to be written by hand. A tool generates these descriptions out of the available RPC documentation.) Each such class contains besides an empty constructor only public instance variables. The instance variables correspond to the parameters of the remote procedure. The instance variable names reflect the parameter names in the remote procedure documentation. A suffix Out marks output parameters. The types of the instance variables correspond to the types of the remote procedure parameters.

In order to call a remote procedure, including all the data fetching and processing that is associated with a call, the reuser sends the message doRPC(...) to the RPC framelet, passing the

remote procedure name as well as the input and output dialog variables as parameters as sketched above.

Based on the remote procedure name, the RPC framelet first searches the class that describes the parameters of the remote procedure, and instantiates this class. The framelet then iterates over the instance variables of this object and assigns to them those values to them which it retrieves from the GUI elements of the input dialog window that have the same name as the parameters in the description object. For this purpose, the framelet iterates over the instance variables of the dialog window. This works fine as the GUI elements of a dialog window manifest in public instance variables of that dialog window object.

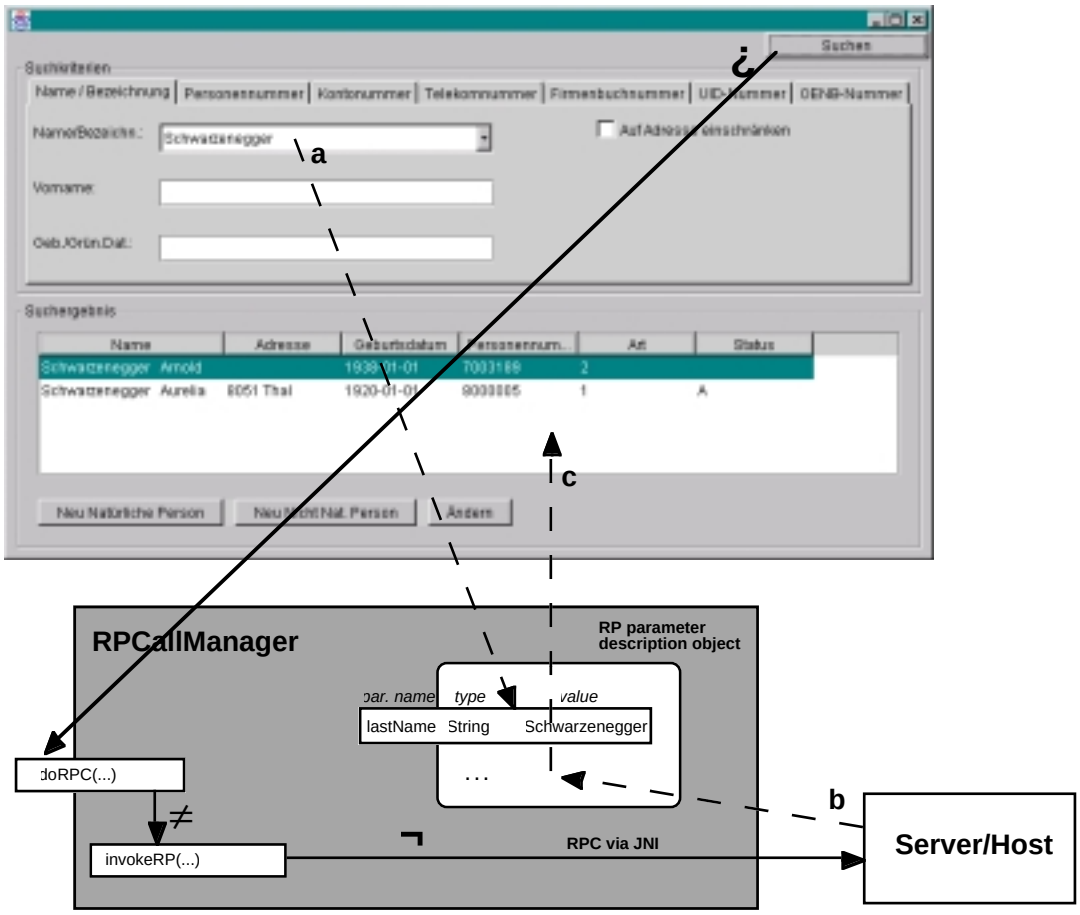

Figure 4 Schematic representation of interactions and data flow in a RPC framelet adaptation.

Figure 4 exemplifies the interaction between various components during the invocation of a remote procedure. The solid lines again depict control flow, whereas the dotted lines represent data flow. Activating the button Search (=“Suchen” in the dialog window with German labels) should imply the invocation of a remote procedure SearchPerson which basically searches all records in a database that correspond to the search parameters (eg, the entered last name). First the method doRPC(...) is called (¿) and receives the following parameters: the name of the remote procedure as a string, and the references to the input and output dialog windows. In this example both refer to the same dialog. The RPC framelet now retrieves the values from the input

dialog window in order to assign these values to the remote procedure parameter description object (a) and calls the remote procedure (≠ and ¬). Note that the name of the GUI element which displays the string Schwarzenegger is not visible in Figure 4. The GUI element has the internal[2] name *lastName* and thus adheres to the naming convention. One instance variable of the remote procedure description object also has the name *lastName*.

The RPC framelet finally processes the results returned from the host or server and assigns the values to the proper instance variables of the remote procedure description object (b). The remote procedure description object provides some additional information how to process the result (a C-array structure) for each remote procedure. This detail will not be discussed in this paper. From there the RPC framelet transfers the data via the meta-information interface and naming convention into the GUI elements of the output dialog window (c).

The source code in Example 1 illustrates how reflection allows the generic implementation of a RPC. The second parameter of this method is the remote procedure description object whose role is explained above.

The classes Class, Method and Field are part of the standard Java library. The first line of method invokeRP(...) stores all instance variables (fields in Java jargon) of the remote procedure description object in an array. Suppose that the object has N instance variables, then the arrays, params, and args have the initial size N=fields.length. The for-loop assigns the particular array component the type (params[i]= fields[i].getType()) and the value of the instance variable (args[i]= fields[i].get(parametersOfRPC)).

The class ListOfRPCs contains all remote procedures as methods. The methods invoke the associated C functions by means of the Java Native Interface (JNI). The statement getMethod(...) returns the Method object that corresponds to the name of the remote procedure. This is the first parameter of method invokeRP(...). The reference to this Method object is stored in the variable RPCmethod. Class Method offers a method invoke(...) to finally carry out the call.

The selected source code illustrates how reflection is useful to decouple the framelet from the specific remote procedure library. The description of a remote procedure in a separate class suffices for a generic implementation of a remote procedure call in the framelet. New or changed remote procedures only require additional or modified descriptions. The RPC framelet itself is not affected.

```
class RPCallManager ... {
    ListOfRPCs rpcList;  // contains all RPCs as methods
    ...
    RPCallManager (...)  {
            ...
    }
    ...
    public void doRPC(String RPCname, Object inDialog, Object outDialog) {
            ...
    }
```

[2] The GUI editor assigns a name to each GUI element. A tool generates Java code which corresponds to the visual/interactive specification of the GUI. In general, a dialog window is represented in one class. The GUI elements contained in a dialog window become instance variables of this class. The GUI element names determine the instance variable names.

```
    protected void invokeRP(String nameOfRPC,
                            Object parametersOfRPC)  {
        Field[] fields = parametersOfRPC.getClass().getDeclaredFields();
        Method RPCmethod = null;              // auxiliary var. for invoking RPC
        Class[] params = new Class[fields.length];
        Object[] args = new Object[fields.length];

        for all params do {
            params[i] = fields[i].getType();  // type of parameter
            try {
                args[i] = fields[i].get(parametersOfRPC);  // par. value
            } catch (IllegalAccessException iae) { ... }
        }
        RPCmethod = rpcList.getClass().getMethod(nameOfRPC, params);
        ... // exception handling
        RPCmethod.invoke(args);
        ... // exception handling
    }
}
```

Example 1 Generic implementation of the remote procedure call.

Overall, the automated configuration of the RPC product line relies solely on meta- information. A method of class Class called newInstance() allows the instantiation of a class whose name is provided as string. Class Class also offers methods for iterating over the instance variables of an object. Both properties together are sufficient for the implementation of the RPC framelet.

Measurements of the run-time overhead of iterating over instance variables showed that the overhead can be neglected. The time for generically assembling a RPC takes between 0.2 and 0.5% of an RPC.

5 Conclusion

We have introduced two basic concepts for extracting reusable elements from legacy systems: framelets and dynamic specialization through reflection. The latter mechanism supports the idea of a framelet by automating part of the specialization work. Neither of these concepts is strictly limited to the OO world, but our discussion and case study have been carried out in the context of OO: this paradigm fits well our purposes through its mechanisms for abstraction, specialization and reflection. To some extent, corresponding mechanisms are provided e.g. by various component technologies (say, COM).

It should be emphasized that dynamically configurable framelets are not only useful for restructuring legacy systems, but they can and should be used as basic architectural units in the design of new systems as well. Since a framelet implements only a restricted functionality, its development is expected to be far less iterative than the development of a typical application framework. Hence, a mature generic software system based on framelets can be developed in essentially shorter time than a conventional framework, yet retaining the applicability of a framework.

The feasibility of framelets may depend on the overall architectural style. It seems that framelets are particularly natural units in a layered architecture where the services required by a layer are implemented by a lower layer. In this case a single layer can be sliced into several framelets. For

each such slice, the interface to the upper layer represents the specialization interface while the interface to the lower layer represents the service interface of the framelet.

Though framework-related design patterns (Gamma et al., 1995; Buschmann et al., 1996) represent architectural knowledge, they are too small to become the foundation of reusable architectural components. Based on the first experience with framelets we argue that framelets might be a pragmatic compromise between design patterns and application frameworks. Framelets might be viewed as the combination of a few design patterns into a reusable architectural building block.

To which degree an application can be based on framelets remains an open question, but we feel that frequently used independent functionalities suitable for framelets can be easily found in many application domains. Future work will focus on the prototypical development of framelet families, on investigation of pragmatic semantic conventions used for the automatic configuration of framelets, and on programming tools supporting the use of framelets.

References

Bass L., Clements P., Kazman R. (1998) *Software Architecture in Practice*. Addison-Wesley 1998.

Bosch, J, Mattsson, M., and Fayad, M. (1998): Framework Problems, Causes, and Solutions, CACM, 1998 (will appear)

Buschmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M. (1996) *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley and Sons

Casais E. (1995): An Experiment in Framework Development. *Theory and Practice of Object Systems* 1, 4(1995), 269-280.

Fayad, M. and Schmidt, D (1997) Object-Oriented Application Frameworks. CACM, Vol. 40, No. 10, October 1997.

Gamma E., Helm R., Johnson R. and Vlissides J. (1995). *Design Patterns—Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley

Pree W. (1996). *Framework Patterns*. New York City: SIGS Books (German translation, 1997: *Komponentenbasierte Softwareentwicklung mit Frameworks*. Heidelberg: dpunkt)

Pree W. and Koskimies K. (1998): Framelets—Small and Loosely Coupled Frameworks. ACM Symposium on Frameworks (will appear)

Sparks S., Benner K., Faris C. (1996): Managing Object-Oriented Framework Reuse. Computer 29,9 (Sept 96), 52-62.