8. Lean Product-Line Architectures for Client-Server Systems — Concepts & Experience

Wolfgang Pree

Summary

This chapter describes the context and software engineering issues of a technology migration project. Significant parts of a conventional client-server system have been replaced by a set of lean product-line architectures implemented in Java. The application of framework and Java technology leads to better modularisation and to improved component reuse. Experience has proven that the focus on small frameworks, called framelets, and on partially self-configuring components are a key factor for rearchitecturing legacy systems. A brief discussion of project management aspects and an evaluation of the applied Java technology rounds out the chapter.

8.1 Introduction

CACS (Component Architectures for Client-Server Systems) is a cooperative project between RACON-Linz Software GmbH, a software company of the Austrian Raiffeisen banking group, and the Software & Web Engineering Group at the University of Constance. CACS began at the end of 1997. The project started by evaluating whether framework and Java technology could be applied in the realm of the RACON-specific client-server architecture[1]. The following aspects had to be considered in particular:

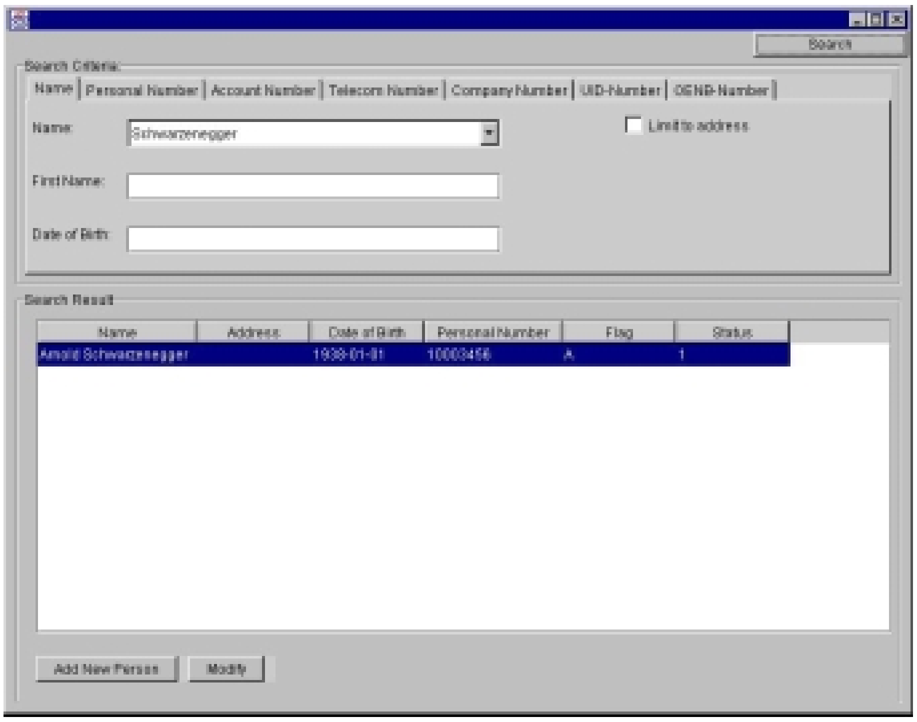
- what are the implications of replacing a 4th generation language (4GL) tool for developing the client system parts with a development environment based on Java? In particular, can the object-oriented approach lead to a significantly better modularisation?
- can Java improve the productivity compared to the 4GL tool?
- what are the qualitative and quantitative differences between a Java and a 4GL solution?
- is the integration of server and host components possible? RACON uses a remote procedure library implemented in C to manipulate data stored on the servers and the mainframe.

In order to evaluate the various aspects of framework and Java technology, we implemented representative system aspects resulting in a CACS prototype. It became clear

[1] RACON-Linz develops three-tier client-server applications for the banking domain, consisting of the host layer (mainframe), the server layer (currently Unix servers; the migration to Windows NT is on the way) and the client side with Windows PCs.

quite early in the project that the available Java technology represented a viable alternative to the Windows 4GL tools in use, and so the focus of CACS shifted to the development of appropriate small product line architectures in Java.

After presenting the characteristics of the CACS prototype's architecture, this chapter discusses how framework technology was applied in this specific setting in order to gain reusable architecture components. Some facts about the project itself and an evaluation of Java technology in the realm of CACS conclude this chapter.



Name: Schwarzenegger

First Name:

Date of Birth:

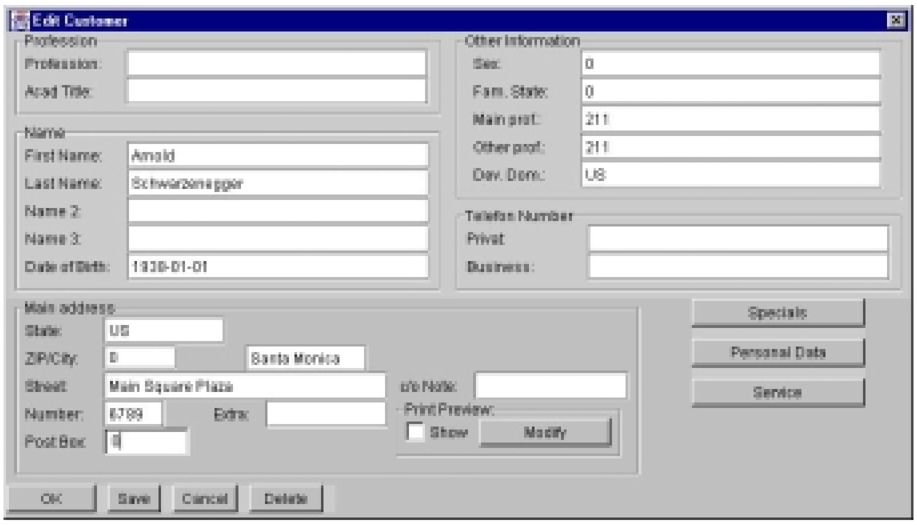
Name	Address	Date of Birth	Personal Number	Flag	Status
Arnold Schwarzenegger		1938-01-01	10003456	A	1

Figure 8.1. Searching for customers with the last name Schwarzenegger

8.2 The CACS System: an End User Perspective

Two screen shots demonstrate the characteristics of the CACS prototype from the perspective of an end user. The overall client system consists of several hundred dialogues analogous to the ones depicted in Figures 8.1 and 8.2. All these dialogues provide quite simple GUI dialogue elements, such as edit text fields, pop-up menus, buttons, and lists. The principal purpose of the dialogues is to enter data for queries on the mainframe/server databases and to display the results.

For example, an end user wants to retrieve information about a bank customer by means of the dialogue in Figure 8.1. The tab control allows the selection of various


Edit Customer

Profession
Profession:
Acad Title:

Name
First Name: Arnold
Last Name: Schwarzenegger
Name 2:
Name 3:
Date of Birth: 1930-01-01

Other Information
Sex: 0
Fam. State: 0
Main prof.: 211
Other prof.: 211
Dev. Dom.: US

Telefon Number
Privat:
Business:

Main address
State: US
ZIP/City: 0 Santa Monica
Street: Main Square Plaza
Number: 6789
Extra:
Post Box: 0
c/o Note:
Print Preview:

Figure 8.2. Dialogue for editing customer data

search criteria such as name, personal identification number, account number, and telephone number. In the case of a name-based search, the end user enters the last name, and/or first name, and/or the birthdate. After pressing the Search button, the list in the lower half of the dialogue displays the search results, i.e. customers with the last name Schwarzenegger.

In order to display or modify the detailed information associated with the customer selected in the list, the end user presses the Modify button in the dialogue in Figure 8.1. This opens the edit dialogue (see Figure 8.2).

8.3 Rearchitecting Client-Server Systems

Many 4GL tools produce a monolithic architecture on the client side. All dialogue windows form one executable which has to be loaded to a client — no matter how small the percentage of required dialogues. From a development point of view, it is hardly possible to package dialogues, or parts of dialogues, into reusable components. From an architectural point of view, the module structure of the client system is quite simple to envision: one dialogue forms one module. In some cases a small group of dialogues might be packaged into one unit. Of course, object-oriented languages such as Java and C++ allow such a modularisation.

A closer look at the resulting module structure of dialogues 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. Naturally, 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 with most dialogues in the real world client-server systems have one or more GUI elements that display items in lists (by means of a GUI component called multi-column grid control; see dialogue in Figure 8.1), interactions associated with lists are also replicated in most dialogues. Pressing a button to add an item opens a dialogue window for entering data. Pressing a button to modify an item also opens a dialogue window and transfers 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 listed items, the dialogue window to display an item and some details such as button labels and the location of buttons for manipulating the list (for example, below the grid control or on the side of it).

Figure 8.3 illustrates schematically the problem of replicated components in the architecture of the client-server system at hand. As proposed by Bass et al. [1], solid arrows represent control flow, dotted arrows depict data flow. Though the size of the replicated components is small (about 200 to 300 lines of source code), they are implemented from scratch several hundred or even thousand times.

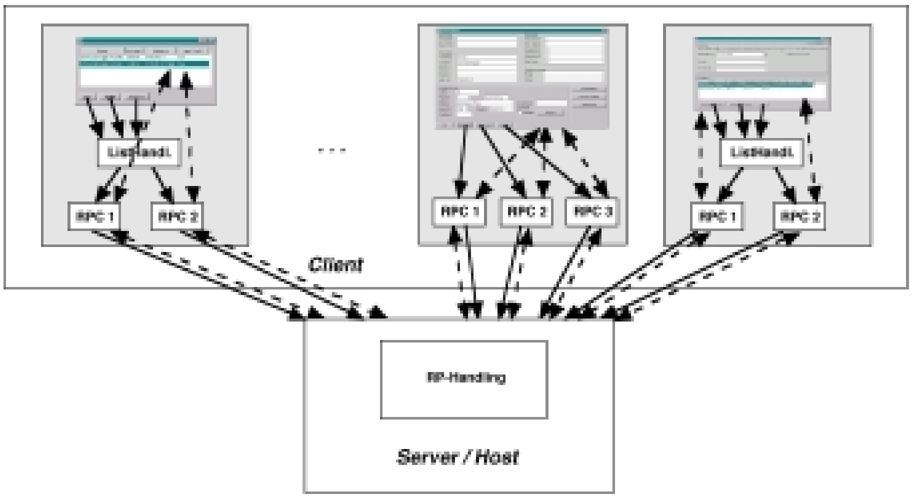


Figure 8.3. Architecture of a client-server system with replicated components

In both cases — remote procedure calls and list handling — the differences between the specific instances are subtle, so a procedure/method with parameters would not suffice. In each of the replicated code fragments a few components interact. Thus framework technology forms a sound basis for generically implementing these system aspects. Figure 8.4 illustrates the idea for rearchitecturing such systems based on small product line architectures. The two components representing the product line architectures are shown as grey ovals.

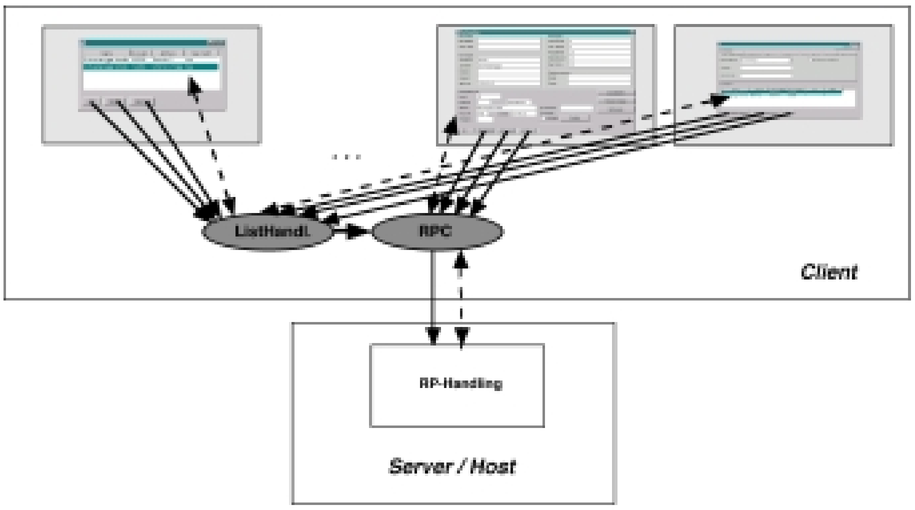


Figure 8.4. Architecture of the client-server system with two small frameworks

8.4 Framework Technology and Reflection as a Basis of Self-Configuring Product-Line Architectures

For a detailed description of the concepts and construction principles of object-oriented frameworks we refer the reader to [5, 6, 2, 3]. The term framework implies no upper or lower limits regarding the number of its components: the same is true for the size of the particular components. Thus frameworks can range from one or a few simple classes to large sets of complex classes. Nevertheless, the tendency in the software community is to develop large frameworks comprising generic solutions for various application domains, such as manufacturing, banking, health care, and process control systems. Sparks et al. [8] discuss typical problems associated with complex frameworks. Not only the design, but also the reuse of such artifacts becomes difficult. The integration of large frameworks forms another challenge.

We argue that the reason for these problems is the conventional idea of a framework as a skeleton of a complex, fully-fledged application. Consequently, a framework becomes a large and tightly coupled collection of classes that breaks sound modularisation 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 specialised 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.

8.4.1 Framelets — Small is Beautiful

Given these problems, we propose a significant downsizing of frameworks and introduce the term framelet [2] to emphasise that it is a small 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 any framework, a framelet can be specialised by subclassing and composition. 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 means of modularising 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. The case studies in Section 8.5 exemplify a framelet family developed in the CACS project.

Frameworks and framelets have in common that they implement flexible object-oriented software 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 [6], are applicable to framelets as well. A framelet retains the Hollywood principle characteristic of white box frameworks: framelets are assumed to be extended with application-specific code called by the framelet. In other words, the term framelet makes explicit that it is a small framework.

8.4.2 Reflection Versus Abstract Classes and Interfaces

A framework consists of specific components that interact with abstract ones; these are called *hot spots* [6]. Usually, the abstract components are defined as abstract classes or (Java) interfaces. The methods of an abstract component express the semantics associated with a particular component. In typed languages this restricts the set of specific components that can be plugged into the framework. Only instances of classes which are compatible with the abstract components can be plugged in. Usually, these are instances of subclasses of the abstract classes or instances of classes that implement a particular interface.

Instead of restricting the range of components with which the framework can interact, we could allow any component to be plugged in the framework. This is done at the programming language level by choosing the most common class, in most libraries called Object, as the required type. Systems that support meta-information allow various operations on any object, such as iterating over the methods, invoking methods, iterating over the instance variables, and getting/setting the values of (accessible) instance variables.

On first consideration, this seems to be less than useful as no semantics are associated with these operations, as opposed to abstract classes or interfaces whose methods define a specific type with an associated behaviour on which the framework developer can rely.

The advantage of such *reflection-based hot spots* is that assets with a little bit of 'intelligence' can be constructed which exhibit self-configuring properties. The framework or 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 components. The sample framelets discussed in the next section apply a very simple mechanism for defining semantics, i.e. naming conventions. Note 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 drawback of giving up strong typing. They introduce equivalence of types on the domain level. For example, the Oberon operating environment designed and implemented by Wirth and Gutknecht [9] heavily relies on generic message handlers which correspond conceptually to reflection-based hot spots.

Naturally, naming conventions are probably one of the most basic means of defining semantics. We are currently investigating more sophisticated ways of pragmatically defining domain-specific semantics based on speech acts [7].

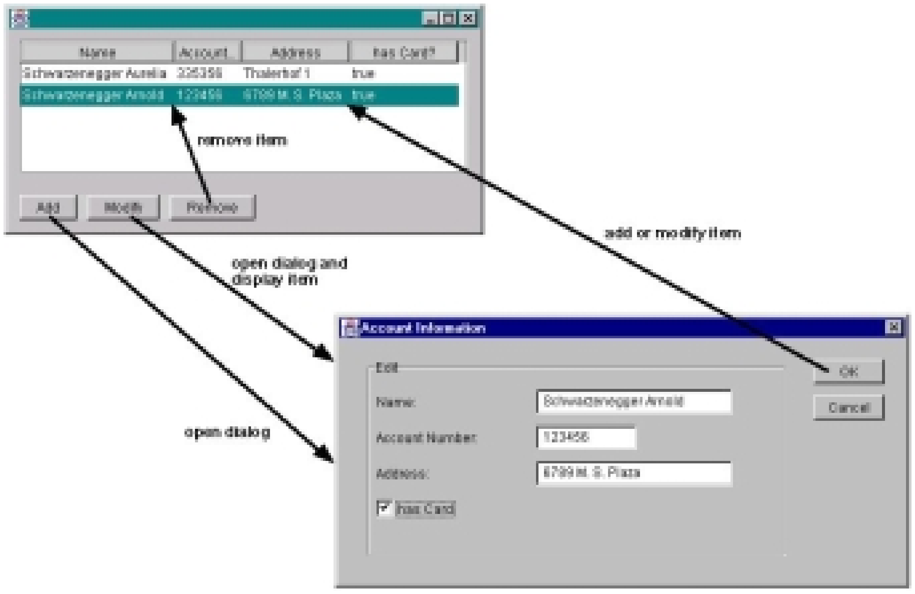
8.5 A Sample Framelet Family

This section sketches the two core framelets developed in the realm of the CACS project, the List Handling framelet and the RPC framelet. They were designed as a framelet family, i.e. as two independent framelets that can nevertheless be easily coupled. The List Handling framelet consists mainly of GUI components, the RPC framelet is a component without GUI representation. Both are provided as Java Beans.

8.5.1 List Handling Framelet

Many dialogues on the client side contain one or more lists of items that have to be displayed in a tabular view. The so-called *grid control* represents the appropriate GUI element for this purpose. A grid control usually comes along with buttons that allow the editing of the list of items, i.e. adding items to the list, removing items from the list, and modifying selected list items. The interactions associated with the corresponding buttons have to be implemented from scratch for each such list, no matter what specific items are handled. For example, the Modify button should only be enabled if an item is selected in the list. The List Handling framelet generically implements these interactions.

Adaptations of the List Handling framelet differ mainly in the dialogue window which displays the attributes of a particular item. Figure 8.5 illustrates a sample adaptation of the framelet. The arrows schematically represent the interactions between the visible framelet components. If the end user presses the Add button or Modify button, the framelet opens the dialogue window for displaying the item's attributes. This is shown in the lower right dialogue in Figure 8.5. Pressing the OK button in this dialogue adds an item to the list or modifies the selected one. The Remove operation



Name	Account	Address	has Card?
Schwarzenegger Aurelia	335356	Thalerhof 1	true
Schwarzenegger Arnold	123456	6789 M. S. Plaza	true

Account Information

Edit

Name: Schwarzenegger Arnold

Account Number: 123456

Address: 6789 M. S. Plaza

☑ has Card

Figure 8.5. Component interactions in a sample adaptation of the List Handling framelet.

(Remove button in the upper left dialogue) deletes the selected list item. The framelet also has to take care that the buttons Modify and Remove are only enabled if an item is selected in the list.

Design and implementation of the list handling framelet The interface of a reusable asset should be designed in as straightforward a way as possible for the user. If the list handling is packaged in a reusable asset, the ideal situation would be that the reuser just provides the following components/parameters:

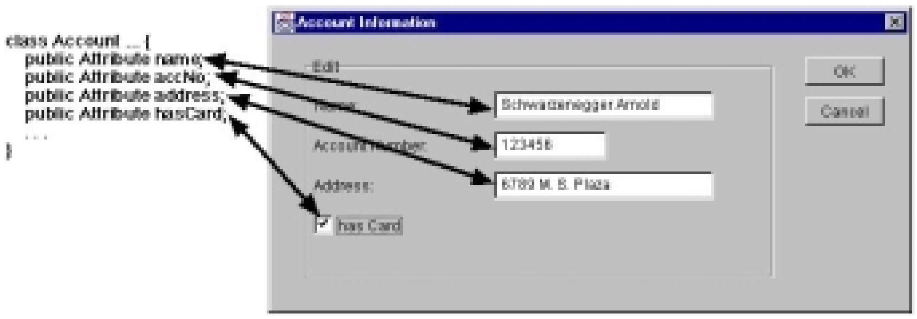
- the dialogue window that displays the item attributes;
- the class which represents a list item;
- if necessary, the names of the remote procedures that have to be invoked for storing and retrieving list items.

The reflection-based hot spots of the List Handling framelet are the attribute dialogue window and the item's class. The framelet automatically transfers the attribute values of an item between the instance variables of any object and the attribute dialogue window based on meta-information and a simple naming convention. The names of the instance variables of the class whose instances represent the list items must have the same name as the GUI elements in the dialogue. For example, a class `Account` has the instance variables `name, accNo, address`, and `hasCard`. The names of the GUI elements in the dialogue window correspond to the instance variable names (see Figure 8.6). Note that the internal names[2] of the GUI elements are not visible in Fig-

[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 dialogue window is

ure 8.5. For example, the GUI element with the internal name `address` displays the string “6789 M.S. Plaza” in the screen shot. The static text “Address:” represents a separate GUI element that does not correspond to an instance variable in class `Account`.

Let us take a look at how the automated transfer takes place from an attribute dialogue to an item. The framelet iterates over the instance variables of an item and assigns to them those values which it retrieves from the GUI elements of the attribute dialogue window that have the same name. For this purpose, the framelet iterates over the instance variables of the dialogue window. The precondition for this is that the GUI elements of a dialogue window manifest in public instance variables of that dialogue window object. An analogous process transfers data in the other direction, i.e. from an item to an attribute dialogue.



```
class Account ... {
   public Attribute name;
   public Attribute accNo;
   public Attribute address;
   public Attribute hasCard;
   ...
}
```

Name: Schwarzenegger Arnold
Account number: 123456
Address: 6789 M. S. Plaza
[✓ has Card]

Figure 8.6. Automated data transfer between list item and dialogue window

The sketched transfer mechanism works independently of the specific item class and attribute dialogue window. In this sense the two reflection-based hot spots — the item and the dialogue — are generically coupled by the framelet according to the simple naming convention.

8.5.2 RPC Framelet

Reusing the RPC framelet should also require minimal effort on the client side. One has to provide the following:

- the name of the remote procedure as a string;
- a reference to an object whose instance variables correspond to the input parameters of the remote procedure. Typically this is a reference to the dialogue window (= input dialogue window) that contains the GUI elements which should become the input parameters of the remote procedure;
- a reference to an object whose instance variables correspond to the output parameters of the remote procedure. In most cases this is a reference to the dialogue window (= output dialogue window) whose GUI elements display the values of the result parameter.

---

represented in one class. The GUI elements contained in a dialogue window become instance variables of this class. The GUI element names determine the instance variable names.

The input dialogue window and the output dialogue window can be identical. Both dialogue windows represent the reflection-based hot spots of the RPC framelet. A naming convention analogous to the one in the List Handling framework allows the automated data transfer between the dialogues and the remote procedure parameters. The GUI elements of the dialogues must have the same names as the remote procedure parameters.

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 do not have to be written by hand. A tool generates these descriptions out of the available remote procedure documentation. Besides, an empty constructor for each such class contains 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.

Figure 8.7 schematically depicts the interactions between the framelet components and the data flow. According to the remote procedure name, the framelet generates the corresponding parameter description object. Relying on the naming convention and meta-information, the framelet then transfers the data from the input dialogue window to the input parameters, calls the remote procedure and assigns the output parameters to the corresponding GUI elements of the output dialogue window. We do not describe the details of interpreting the return parameter which is provided as a C-array. This can also be done generically for all remote procedures based on a return parameter description.

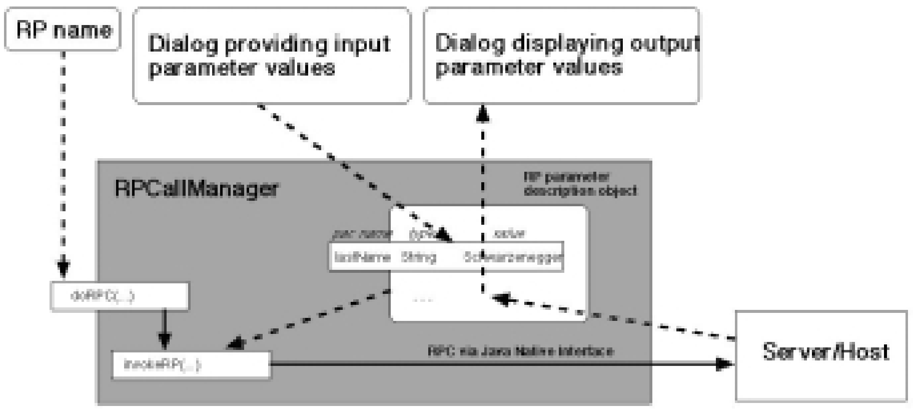

RP name
Dialog providing input parameter values
Dialog displaying output parameter values
RPCallManager
RP parameter description object
par. name
type
value
lastName
String
Schwarzenegger
...
doRPC(...)
invokeRP(...)
RPC via Java Native Interface
Server/Host

Figure 8.7. Schematic representation of the internal structure and working of the RPC framelet

8.5.3 Framelet Coupling

Both framelets can be used on their own or together — they form a very small framelet family. The previous section sketches the stand-alone usage of the RPC framelet. The framelet is directly coupled with the dialogue(s) that contains the input/output parameter values. Otherwise, the List Handling framelet invokes remote procedures and should reuse the RPC framelet. In this case the reflection-based hot spots allow such flexible reuse. The RPC framelet couples itself with any object that adheres to the naming conventions. The List Handling framelet just has to pass the object(s) whose instance variables represent the input and output parameters of the remote procedure. This can, for example, be an instance of the item's class or a dialogue, depending on the specific context.

8.6 Discussion and Conclusions

The RACON-Linz management clearly positioned the CACS project project as a foundation for technology migration, and thus separated it from day-to-day business. Five people worked on the prototype for 8 months, two people at RACON-Linz and three at the Software & Web Engineering Group at the University of Constance. All members of the project team had in-depth knowledge of object and Java technology including experience in developing and adapting frameworks. The current state of Java technology is sufficient for fulfilling the specified development requirements. Within the CACS project around ten framelets have been developed so far. The List handling and RPC framelets were reused in building the CACS prototype.

8.6.1 Java Evaluation

The CACS prototype is based on Java 2, i.e. the Java Development Kit (JDK) version 1.2. It was developed with Borland/Inprise JBuilder 1.0, later 2.0 on Windows NT. The CACS prototype is a just-in-time compiled stand-alone application. The GUI was developed originally with the Abstract Windowing Toolkit (AWT) and later with the Swing library. The Java Native Interface (JNI) forms the basis of the integration of the RPC library which is a C library. The resulting framelets are provided as Java Beans. JavaDoc was used for documentation. JBuilder together with the Java GUI libraries offer — like other Java development environments — a viable alternative to 4GL tools regarding the visual/interactive editing of dialogues.

From the language point of view, Java provides the meta-information support necessary for the configuration capabilities implemented in the List Handling and RPC framelets. The small number of language features in Java has a positive effect on the quality of the program source code, though this impact is difficult to measure.

8.6.2 Quantitative Data

The CACS prototype executes on Windows PC clients. The execution speed of just-in-time compiled stand-alone applications is adequate on the target platform, i.e. on

PCs with processors running at $\geq$ 100 MHz. The time-critical queries take place on the servers or on the host invoked via the RPC interface. Measurements of the run-time overhead of iterating over instance variables showed that overhead introduced through reflection can be ignored. The time for generically assembling a RPC takes between 0.2 and 0.5% of a remote procedure call.

The CACS prototype can be described quantitatively as follows:

number of classes	25 (13 out of the 25 were generated by the GUI editing tool)
number of interfaces	2
number of lines of code (LOC)	ca. 2700
depth of class hierarchy of the newly developed classes	2
number of reused Java components (GUI)	10
number of reused framelet components	2

8.6.3 Software Engineering Challenges

The focus on the development of small product line architectures allows the opening up of a significant reuse potential. In particular, framelets and dynamic specialisation through reflection turned out to be adequate means for extracting reusable elements from legacy systems.

On the other hand, the develoment of reusable assets implies organisational changes. The Expert Services Model, as described, for example, by Goldberg and Rubin [4], seems to be suitable for promoting the development and evolution of domain-specific framelets. In essence, the Expert Services Model comprises a reuse competence centre, which is staffed by an independent reuse team. Two members of this reuse team are responsible at RACON-Linz for identifying, acquiring, certifying and storing the reusable components in a shared library. This organisational measurement will be evaluated in the future and, if necessary, adjusted to the specific needs.

In order to come closer to the vision of partially self-configuring components, pragmatic ways of describing component semantics still have to be found; the applied naming conventions form a starting point. We feel that domain-specific vocabulary and quite restricted semantics, instead of general purpose formalisms, could lead to semi-automated component configuration workbenches that can be useful in practice.

References

1. Bass L, Clements P, Kazman R. Software Architecture in Practice, Addison-Wesley, Reading, Massachusetts, 1998
2. Fayad M, Schmidt D, Johnson R. Building Application Frameworks: Object-Oriented Foundations of Framework Design, John Wiley & Sons, New York City, 1999

3. Fayad M, Schmidt D. Object-Oriented Application Frameworks. Communications of the ACM 1997;Vol. 40, No. 10
4. Goldberg A, Rubin K. Succeeding with Objects — Decision Frameworks for Project Management, Addison-Wesley, Reading, Massachusetts, 1995
5. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns — Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts, 1995
6. Pree W. Design Patterns for Object-Oriented Software Development, Addison-Wesley, Reading, Massachusetts, 1995
7. Shoham Y. An Overview of Agent-Oriented Programming. In: Bradshaw J (ed) Software Agents, Cambridge, Massachusetts, AAAI Press/The MIT Press, 1997, pp 271–290
8. Sparks S, Benner K, Faris C. Managing Object-Oriented Framework Reuse. Computer 1996;Vol. 29, No. 9
9. Wirth N, Gutknecht J. Project Oberon: The Design of an Operating System and Compiler, Addison-Wesley, Reading, Massachusetts, 1992