Self-Configuring Components for Client-/Server-Applications

Wolfgang Pree, Egbert Althammer
Software Engineering Group
University of Constance
D-78457 Constance, Germany
{pree, althammer}@acm.org

Hermann Sikora
GRZ/RACON Software GmbH
Goethestr. 80
A-4020 Linz, Austria
sikora@grz.at

Abstract

A mechanistic view of software component assembly implies exact matching and fitting of the particular components. We argue that components for large-scale software construction should have automatic configuration capabilities in order to significantly enhance their reusability and maintainabilty. The paper sketches a pragmatic approach for implementing self-configuring components, relying on framework technology, meta-information and naming conventions. A case-study drawn from the client-/ server-domain illustrates the concepts.

1 Beyond Open Implementation

Gregor Kiczales [1] discusses the problems of black-box abstraction: The implementation details of a module/component are hidden. In many cases, clients are affected by the implementation and its specific restrictions so that programmers have to "code around" the component implementation. Thus implementations should be opened up to allow clients more control over these issues. Figure 1 illustrates the basic concept of an open component implementation, which provides two interfaces, a primary interface and a meta-interface. A client requests the functionality of a component through the primary interface. The meta-interface allows the parameterization of the various component implementation strategies.

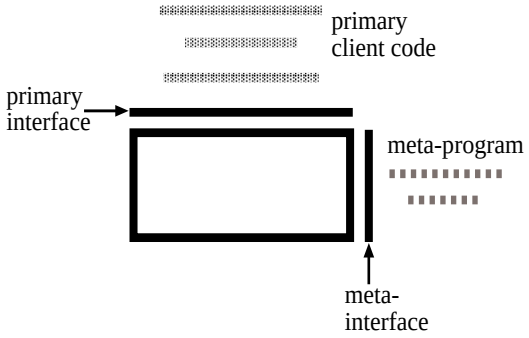

Figure 1 A component based on the open implementation principle (adapted from [1])

Adjustment through computational reflection represents a visionary aspect of open implementation interfaces: components should configure themselves by providing interfaces for examining and adapting themselves. This focuses on the optimization of single component implementations.

Though a somehow automated optimization of component implementations can be helpful, we view the assembly of components as a step beyond the open implementation idea. The long-term vision is automated component assembly in the large scale: Components advertise their capabilities, and engage in mutual interactions and negotiations. Components would inspect the meta-descriptions and recognize that some functionality would be appropriate for a specific task.

The vision sketched above is very difficult to realize. Nevertheless, the application of well-known concepts—framework technology, computational reflection, and naming conventions—allows the design and implementation of self-configuring components, which form a small step towards reaching the much more ambitious goal. The subsequent section discusses the role of framelets, a deviate of frameworks, as basis of self-configuring components. A case-study corroborates that the reuse of self-configuring components in real-world applications can lead to significantly leaner software systems.

2 Framelets as architectural building blocks

Not only the design of complex frameworks is hard, but also the reuse of such artifacts (see, for example, [1, 3, 5]). 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. Furthermore, the internal working of different frameworks is usually not compatible so that two or more frameworks can hardly be combined.

We argue that the reason for these problems is the conventional idea of a framework as a skeleton of a complex, full-fledged application. Consequently, a framework becomes a large and tightly coupled collection of classes that breaks sound modularization principles and is difficult to combine with other frameworks. Inheritance interfaces and various hidden logical dependencies cannot be managed by application programmers.

As alternative we propose *framelets* as small architectural building blocks that can be easily understood, modified and combined. Note that not the construction principles of frameworks form a problem, but the granularity of systems where they are applied. The starting point to tackle these problems is the following: If one takes a look at the source code of various software systems, numerous small aspects are implemented several times in a similar way. Thus, framework concepts can be applied to the construction of such small, flexible reusable assets. We call these building blocks framelets. In contrast to a conventional framework, a framelet

- Is small in size (< 10 classes),
- Does not assume main control of an application,
- Has a clearly defined simple interface.

We base our research of self-configuring components on framelets as small, flexible architectural building blocks. The vision is to have a family of related, partially self-configuring framelets for a domain area representing an alternative to complex frameworks. We view framelets as a kind of modularization means of frameworks. In large scale, an application is constructed using framelets as black-box components, in small scale each framelet is a tiny white-box framework.

Commonalties between frameworks and framelets

Frameworks and framelets have in common that they implement flexible object-oriented software architectures. (Note that the definition of the term framework does not imply any size. Introducing the term framelet allows us to discern between complex frameworks and small ones which have the characteristics stated above.) For this purpose frameworks and framelets rely on the constructs provided by object-oriented programming languages. The few essential framework construction principles, as described, for example, by Pree [4], are applicable to both. A framelet retains the Hollywood principle characteristic to white-box frameworks: Framelets are assumed to be extended with application-specific code called by the framelet.

3 Combination of framelets, meta-information, and naming conventions—a case-study

The goal of the case study is to demonstrate how framelets help to avoid the reimplementation of certain aspects of a software system again and again from scratch. We applied the concept to a typical client-/server setting in a bank, using Java as implementation language. The architectural design of the client-/server-system relies on remote procedure calls (RPCs). Around 50 procedures/functions are provided as a C-library for

transferring data between the clients and the other tiers of the system. The code associated with a particular RPC implies tedious programming work for handling parameter value transfers. For example, the return parameter types are C-style arrays which have to be properly processed. Investigating the code structure reveals that the parameter exchange is similar for all RPCs. Nevertheless the RPCs are too diverse to come up with a simple procedure/function. The self-configuring RPC framelet discussed below does the job.

RPC framelet design

One important design goal of a reusable asset is to bother its reuser as little as possible. This means in case of the RPC framelet that the programmer who reuses it has only to specify the RPC name. The parameter list should be assembled almost automatically. In other words the RPC framelet component should provide some self-configuring properties. In our case-study the source of the parameter values are typically GUI dialog elements. Thus, the RPC framelet should automatically manage the transfer of data between the GUI items and the parameter list of the RPC. How could this be accomplished?

The basic idea to automate the data transfer is to establish a simple naming convention: the names of the GUI elements have to be identical with the parameters of the RPC.

Generic RPC implementation

The RPC framelet relies on a description of the parameters of a particular RPC. For each parameter the framelet has to know its type and whether it is an input or output parameter. As mentioned above, the remote procedures are implemented as C functions. A separate structured description, available as plain text file, provides the necessary information for each remote procedure. Based on these descriptions, the RPC framelet manages the parameter value transfer as discussed below.

Example 1 shows the generic implementation of a RPC. The first parameter is the name of the remote procedure; the second parameter is an object whose instance variables describe the RPC parameters and their values. This object is automatically constructed out of the structured parameter description of the corresponding RPC.

```
public void invokeRPC(String nameOfRPC,
                      Object parametersOfRPC) {
  Field[] fields =
        parametersOfRPC.getClass().getDeclaredFields();
  Method RPCmethod = null;  // auxiliary var. for invoking RPC
  ListOfRPCs rpcList = new ListOfRPCs(); // contains a list of
                                         // all RPCs
  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); // value of
                                                    // parameter
        } catch (IllegalAccessException iae) {}
  }
  RPCmethod = rpcList.getClass().getMethod(nameOfRPC,
                                  params);
  ... // exception handling
  RPCmethod .invoke(args);
  ... // exception handling
}
```

Example 1. Generic implementation of a RPC.

Data transfer via naming conventions and computational reflection

The reuser of the RPC framelet just has to make sure that the GUI item names are identical with the names of the RPC parameters in the RPC description. A generic matcher object, which is part of the RPC framelet, automatically generates an object which contains the description and values of the RPC parameters, and transfers the data from the dialog items to the corresponding instance variables. The output data of the RPC are shuffled in an analogous way back to the GUI items where they should be displayed.

The automation of the data transfer also relies on meta information: When transferring data from GUI items to a parameter container object, the matcher iterates over the instance variables of the corresponding class, gets the name, type and value of each instance variable and looks for the corresponding GUI item. The same is true for the transfer in the other direction, that is, from the RPC back to the GUI items.

Reuser's perspective

Note that a reuser of the RPC framelet even does not call the invokeRPC() method shown in Example 1, which would require the instantiation and initialization of a specific parameter description class. Instead, the reuser invokes the method constructRPC(...) of the RPC framelet. This method has the following parameters: the name of a remote procedure and the references to the dialogs where the data are entered and the results are displayed. The rest happens automatically via the RPC framelet's self-configuring capabilities.

4 Towards automated component assembly

The set of components in the case study comprise a very small component, ie, the black-box RPC framelet, which is designed for an automatic integration with the bank-specific GUI dialog components. Though this reduces significantly the source code size compared to the original conventional implementation, one can argue that there is no automation involved at all. The case study just corroborates the following: Framelets can in fact be practical building blocks for various kinds of applications. The additional application of computational reflection together with simple naming conventions shows the potential of partially self-configuring components. As clients do not have to figure out how to exactly glue together these assets, reuse can be significantly simplified. But the introduction of naming conventions is far away from components advertising their capabilities, and engaging in mutual interactions and negotiations.

Domain-specific automation

The case study indicates that pragmatic steps towards automated component assembly require restrictions of the domain area wherein automated component assembly can take place. We are currently investigating the definition of a simple negotiation protocol where components of the bank-specific GUI dialog framelet family and the RPC framelet can negotiate configurations based on advertised services. The semantics of the negotiation language and protocol is necessarily tightly coupled with the bank-specific client-/server-setting. For example, the protocol assumes knowledge about specific GUI elements such as account number entry fields.

Besides negotiation protocols, our future work will focus on the prototypical development of framelet families and on an evaluation of the coupling mechanisms of such components. In the long term, it might be possible to come up with crisp, but domain-specific meta-descriptions of large-scale components so that more advanced computational reflection mechanisms allow sophisticated automated component configuration.

Besides providing mechanisms for automated component assembly, various questions have to be clarified. For example, how can the reliability of automatically assembled systems be guaranteed? Further, it is unclear how to structure and document systems that consist of numerous "intelligent" components.

References

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

[2] Kiczales, G. (1996) Beyond the Black Box: Open Implementation. IEEE Software, Vol. 13, No. 1, January 1996.

[3] Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J., Schmucker K. (1995): *Object-Oriented Application Frameworks*. Manning Publications/Prentice Hall.

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

[5] Sparks S., Benner K., Faris C. (1996): Managing Object-Oriented Framework Reuse. Computer 29,9; September 96.