Framelets—Small and Loosely Coupled Frameworks

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

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

Abstract. Not only the design of complex frameworks is hard, but also the reuse of such artefacts. Furthermore, the internal working of different frameworks is usually not compatible so that two or more frameworks can hardly be combined. As alternative we propose framelets as small architectural building blocks that can be easily understood, modified and combined. The paper first discusses the concepts underlying framelets and presents a sample framelet implemented in Java.

1 Why small is beautiful

Although object-oriented frameworks are currently one of the most promising approaches to large-scale reuse of software, they have significant problems reported by several authors (e.g., Bosch et al., 1998; Casais, 1995; Fayad and Schmidt, 1997; Sparks et al., 1996; Johnson, 1992; Lewis, 1995):

- The design of a framework 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.
- The reusing 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.

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 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.

In other words, not the construction principles of frameworks form a problem, but the granularity of systems where they are applied to. 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.

The vision is to have a family of related framelets for a domain area representing an alternative to complex frameworks. Thus 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. This issue will be discussed in more detail below.

Commonalities between frameworks and framelets

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 (1996), are applicable to framelets as well. 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.

To illustrate the notion of a framelet we discuss one representative example taken from the GUI-domain and outline some of the implications of this approach. Naturally, framelets are not restricted to the GUI domain.

2 An example: a list box framelet

GUIs of many applications provide numerous list boxes together with buttons to add items to the list box, and to modify and remove them. Thus these GUI elements and their interactions have to be implemented again and again. The associated programming task is a typical example of a piece of programming work that can easily be packaged into a small self-contained framework we call list box framelet.

Specializations of the list box framelet differ only in the dialog box used for modifying an item. Figure 1 schematically illustrates a sample specialization of the list box framelet. The arrows roughly indicate the interactions between the visible framework components: When the end user presses the Add or Modify buttons the framelet pops up the specific dialog box. In case of pressing the Modify button the dialog box fields contain strings that correspond to the selected item in the list box. Pressing the OK button in the dialog box adds an item to the list or changes an item. Pressing the Remove button removes the selected item from the list. The framelet has to take care that the Modify and Remove buttons can only be pressed if an item is selected.

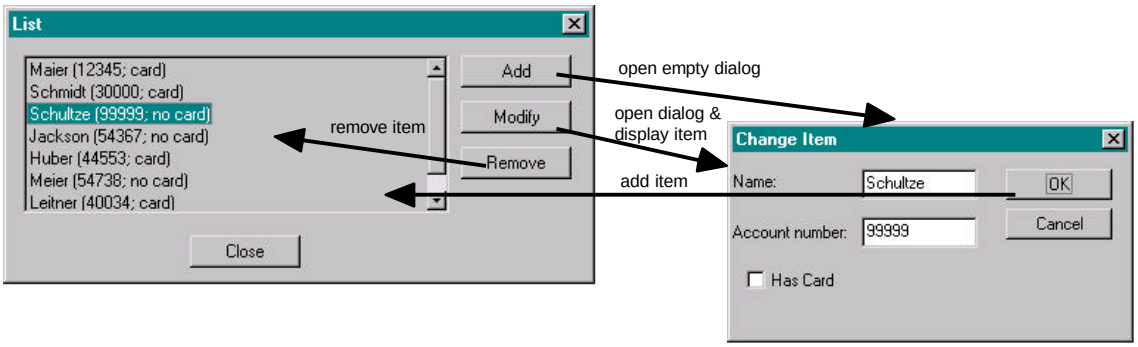

Figure 1 A sample specialization (account information) of the list box framelet

2.1 Design and implementation considerations

One important design goal of a framelet is to bother the reuser as little as possible. This means in case of the list box framelet that the programmer who reuses it has to implement the dialog box to display the data of an item in the list box. Furthermore, the framelet has to know the type of the listed items. Ideally, the framelet manages automatically the transfer of data between the instance variables of an item object and the dialog which displays them. How could this be accomplished?

The basic idea to automate the data transfer is to establish a simple naming convention: The names of the instance variables of the class representing an item are identical with the ones in the item dialog for naming the corresponding GUI elements. According to the screenshots in Figures 1 and 2, a class Account could correspond to the type of listed items. One of Account's instance variables is accountNo. Thus, the edit field in the item dialog also has the name accountNo. (Note that this name is not visible in Figure 2. Figure 2 shows only the label "Account number:", not the name of the corresponding edit field, which displays the string "99999".)

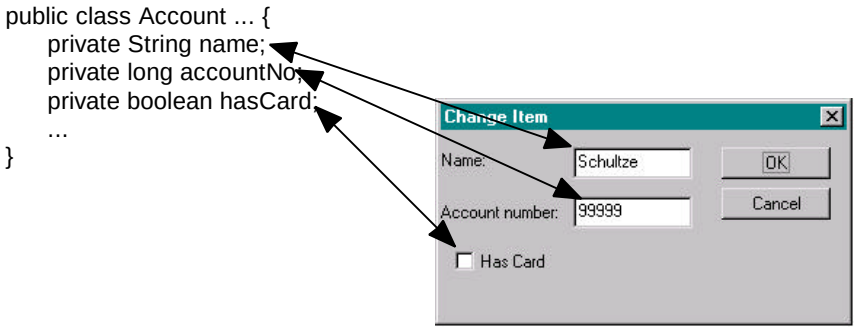

Figure 2 Automated data transfer between list item and dialog box.

One component of the list box framelet, a Matcher object, manages the data transfer in both directions, from an item to the item dialog and vice versa. Figure 2 schematically illustrates the purpose of a matcher. In order to automate the data transfer, the implementation of class Matcher has to rely on meta-information: When transfering data from an item to the item dialog, the matcher iterates over the instance variables of the item class, gets the type and value of each instance variable and looks for the

corresponding GUI element in the item dialog. (Note that the class implementing the dialog has instance variables corresponding to the GUI elements. So the matcher, once knowing the name of an instance variable in the item class searches for the same instance variable in the item dialog.) The transfer in the other direction, that is, from the item dialog to the item works in an analogous way.

The other components of the list box framelet are the LBoxGroup class and the interface DataDisplay. LBoxGroup implements the GUI representation (List component together with the buttons Add, Modify and Remove) and the interactions with the item dialog (see Figure 1). The constructor of LBoxGroup requires just the name of the class that represents the items as String, eg, "Account", a data structure that contains the items, and a dialog of type DataDisplay. The class which corresponds to the item dialog has to implement the interface DataDisplay. This interface offers only one method that reports whether the OK button was pressed or not. If the OK button was pressed by the end user, an item has to be added to the data structure representing the listed items.

2.2 Framelet characteristics and framelet reuse

Remember the characteristics of framelets summarized in section 1. The list box framelet is a representative example of such an artefact:

- The list box framelet does not assume main control of an application. Nevertheless, it relies on the Hollywood principle of white-box architectures. The extension of the list box framelet requires the programmer to provide a class that corresponds to the item dialog and which implements the interface DataDisplay. This is the white-box aspect of the list box framelet.
- The list box framelet is small in size. It consists of the classes Matcher, LBoxGroup, and the interface DataDisplay.
- The interface of the list box framelet is simple and clearly defined. The programmer who reuses this framelet just passes three parameters to the constructor of LBoxGroup as sketched above. The reuser can obtain the collection of list items by invoking the method getItems() of LBoxGroup.

  It can be specified whether a simple list box or one with a tabular grid view is used. This represents a black-box aspect of the framelet.

2.3 Framelet families for client-/server-architectures

Remember the vision to have a family of related framelets for a domain area. The list box framelet can be part of a family for building the client side of applications. Other framelets in this family could be more domain dependent and form groups of GUI elements such as the address framelet for entering and checking a person’s address information.

The GUI framelets can be combined with framelets that automate the construction of remote procedure calls. The coupling could, for example, be based on naming conventions and reflection analogous to the design underlying the Matcher component of the list box framelet.

3 Implications and outlook

We strongly advocate architectures based on small flexible units with lean interfaces. This kind of combination of white-box and black-box architecture aspects appears to

solve the well-known problems of complex application frameworks. The example discussed above demonstrates that such units, framelets, can be practical building blocks for various kinds of applications.

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. For example, the list box framelet is based on a combination of the Observer pattern and the Bridge pattern according to the naming conventions of the Gang-Of-Four design pattern catalog (Gamma et al., 1995). The update mechanism of list items corresponds to the Observer design. The Bridge pattern was applied to design the black-box configuration of the LBoxGroup, that is, the plugging-in of specific list boxes. The generic design involving Matcher and DataDisplay is also relying on the Bridge pattern.

To which degree an application can be based entirely on framelets remains an open question, but we feel that similar small, relatively independent functionalities can be easily found. Future work will focus on the prototypical development of framelet families and on an evaluation of the coupling mechanisms of framelets.

References

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

Johnson R. (1992): Documenting Frameworks using Patterns. In: *OOPSLA '92*, Sigplan Notices27,10 (Oct 92), 63-76.

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.

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

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