Computer Science Journal of Moldova, vol.**15**, no.1(43), 2007

Aspect oriented programming and component assembly

A. Colesnicov L. Malahova

Abstract

The article describes an attempt to use the aspect oriented programming for construction of a graphical user interface from components. Aspects were applied to produce the glue code. Previously some doubts were expressed on possibility of such usage.

1 Introduction

The main efforts in development of complicated software products for scientific applications are applied to the creation of their computational engines that solve their target problems. Supplying a modern graphical user interface (GUI) for the resulting system is a different task that usually meets a shortage of resources. Our own experience with the Bergman symbolic computation system (SCS) and review of other systems illustrate this [1].

We investigate in this article a possibility to create a GUI for a given SCS semi-automatically from a set of ready-made components. The modern techniques of component programming (CP) give us a lot of useful approaches, e.g.: conceptions of black, grey, and white boxes; the notion of the *glue code* (the additional code that is necessary to assemble components together); lists of requirements for components, etc. Nevertheless, we found the existing CP frameworks not fully suitable: they provide less than we need in the automation of component assembly, and they provide much more than we need being oriented mainly towards distributed applications.

Aspect oriented programming (AOP) is a technique to add a new behavior to an existing program without changing its sources and even binaries. It is mostly used to handle cross-cutting concerns like logging or debugging. E.g., we need to add almost the same code in regularly selected places of the program to trace it. AOP concentrates templates of additional code and insertion points in *aspects.* Aspects are compiled separately, and the *code weaving* is performed during the execution of the program.

To apply AOP for the semi-automated assembly of a GUI from components, we noted that the glue code is regular and repeating, and that it can be generated from a formal description of the GUI. With AOP, we use an unchanged GUI template and unchanged components, and generate only aspects containing the glue code. We have checked this idea by implementing it.

Application of the AOP techniques to the CP was claimed in 1999 [2] but was not developed further that time, may be because that solution had supposed language extensions. Moreover, in 2003 C. Perez had published a note in his blog[1] under the title "Do aspects supersede components" that motivates the impossibility of such application. Our article describes a successful experiment in usage of the AOP for component assembly resolving therefore these doubts.

The article begins with a short review (Sec. 2–3) of existing techniques and frameworks of CP. See [3,4] for more details. Sec. 4 briefly describes the AOP. We describe in Sec. 5 our implementation of the idea formulated above. The next Sec. 6 discusses requirements to components. The article terminates with conclusions (Sec. 7) and acknowledgements.

[1] http://www.manageability.org/blog/archive/20030604%23do_aspects_supercede_components

2 Component programming: black-box techniques

Techniques of component adaptation may be roughly classified as *white-box, grey-box,* and *black-box* [5, p. 2–6].

At *white-box* reuse we actually need the full knowledge of component's internals. The most primitive kind of such reuse is the *copy-and-paste* technique known also as the *code scavenging*. Another technique is the *inheritance* usual at object-oriented programming. We meet here at the first time the important notion of the *glue code*, i.e., the code that assembles components together. At the inheritance, the glue code is represented by new methods in the subclasses of the original component.

Grey-boxes were thoroughly discussed in [5]. The following example is adapted from there: let we have a black-box component *Sort* that takes its input and produces the sorted output. We can replace this component. Suppose the initial component used the *Insertion sort* that is stable, i.e., keeps order of records with equal keys. We note this behavior during experimentation and use it in our programming of other parts of our system. Then we replace the component by another one that implements *Tree sort*. This algorithm is unstable, and we are in the wrong side. Therefore some knowledge of this component's internals is necessary. With *Sort*, we can return to the black-box model including stability in its output specifications but there are another situations where this is impossible, e.g., when a component depends on an external service [5, p. 3–5]. Grey boxes are used in most modern technologies.

Black-boxes hide all their internals; only input and output specifications are exposed. Due to their simplicity, this is the most comfortable model for assembly. There are many techniques to manipulate them. We present below several CP technologies based on black-box model.

2.1 Wrapping

This technique is also named *containment* when referred in COM technology (see 3.3 below). A *wrapper* can be defined as a container object that encapsulates a given black-box component and intercepts all its input and output. The simplest wrapper adapts the containing object's interface. More complicated wrappers can restrict or extend the functionality of their containing objects.

Wrappers perform additional calls during the interface tunnelling and can therefore lose in performance.

2.2 Superimposition

Superimposition was introduced by J. Bosch [6]. The component is included inside one ore more *layers* that intercept all messages to and from it. Each layer can convert a message in a passive object, analyze its contents and react in correspondence. Then some additional behavior is superimposed over the initial component's behavior.

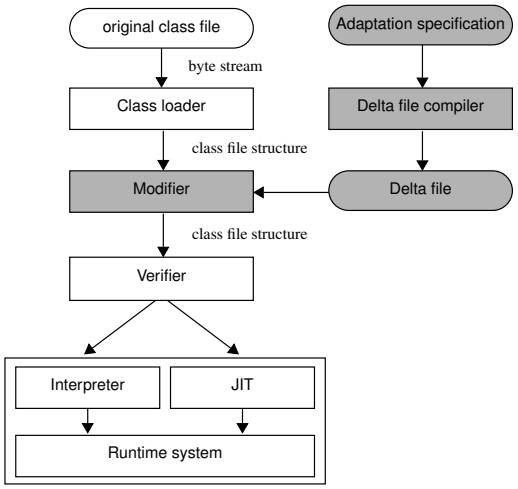

Figure 1. Binary component adaptation

2.3 Active interfaces and binary component adaptation

Active interfaces were proposed by G. Heineman [7]. An active interface decides whether to take action when a method is called. All interface requests pass two phases: the "before-phase" before the component performs any steps towards executing the request, and the "after-phase" when the component has completed all execution steps for the request. The active interface does not adapt component's behavior; instead, it enforces the necessary behavior at run-time.

Binary component adaptation [8] changes compiled (binary) component while it is loaded and was applied to Java. It is illustrated by Fig. 1 taken from [8, p. 2]. We see that *adaptation specification* is prepared separately and compiled separately by a *delta file compiler* to a *delta file*. A *modifier* is a single additional part of the Java virtual machine weaving the delta file with the original class.

Both these techniques were superseded by the *aspect oriented programming* (see Sec. 4 below).

3 Component programming: industrial component models

Attempts to define a set of standards for component implementation, customization, and composition led to such industrial component models as CORBA/CCM, JavaBeabs/EJB, COM, .NET.

3.1 CORBA/CCM

Common Object Request Broker Architecture (CORBA) and *CORBA Component Model* (CCM) were developed by OMG[2]. CORBA is an infrastructure that provides communication of distributed objects or components. The Object Request Broker (ORB) supports communication of components independently of their platforms or the methods of their implementation. All communicating components should be registered in an implementation repository. The components (objects)

[2]Object Management Group, http://www.omg.com

are represented by their *stubs* and *skeletons* that are their platform and language independent abstractions. To create stubs and skeletons, a specialized language called IDL (Interface Definition Language) is used. Instances of stubs look like local objects and accept method invocations from other objects. The actual target objects can be located elsewhere in the network and get these invocations through skeletons. Skeletons handle argument arrangement, actual method invocations and rearrangement of resulting values that are passed back to stubs.

3.2 JavaBeans/EJB

JavaBeans is a technology that is implemented for Java and permits to produce visual (graphical) components that are platform-independent, and reusable.

EJB are *Enterprize JavaBeans* introduced by SUN[3]. EJB are non-visual components of two kinds, *session beans* and *entity beans*. Session beans provide the communication of a client and a database. Entity beans represent the data from the databases and provide methods to manipulate these data. A session bean exists while a connection exists between a client and a database. An entity bean exists while the corresponding data exist in a database.

The EJB model allows the developer to implement business logic of applications and do not go into the things like transactions or security.

3.3 COM

Microsoft's *Common Object Model* (COM) defines a binary structure for interfaces between a COM objects (COM-compliant components) and their clients. There is a standard way to lay out virtual function tables (*vtables*) in memory, and a standard way to call functions through vtables. Components can be implemented in different languages. There exists Microsoft's IDL to define interfaces. Each object can provide several services. All services are registered in a system

[3] http://java.sun.com/developer/onlineTraining/Beans/index.html

registry using a *global unique identifier* (GUID). New implementations of existing objects use new GUIDs.

COM provides two methods of binary reuse, *containment* and *aggregation*. The first one is in fact wrapping: the object intercepts method calls and forwards them to its internal objects. In aggregation, a COM object exposes the services of another object as its own services.

3.4 .NET

The *.NET framework* (now version 3.0; regrettably, different versions are not compatible) is the latest platform from Microsoft that delivers components' services through Internet. A .NET application is composed of *assemblies*. An assembly contains compiled code and metadata. The manifest is included with each assembly and contains the assembly name, its version, the list of files, the list of dependencies, and the list of exported features.

4 Aspect-oriented programming

Aspect-oriented programming (AOP) generalizes, systematizes, and formalizes the *code weaving*. The code weaving was used in many circumstances like logging, tracing, debugging, security checking, etc. An example of early (1989–1990) use of the code weaving at the implementation of a debugger can be found in [9].

In AOP, the existing code is extended by *aspects*. An aspect contains *pointcuts* and *advices*. A pointcut is a template that defines *join points*. Each time the program execution passes a join point, the corresponding advice is executed. The advice code seems to be weaved with the original program code at the join point.

The *source code weaving* is a single possible solution in languages like C++. On the contrary, the Java binary code is well-defined and permits the *binary code weaving*. The AspectJ[4] implementation of

[4] http://www.eclipse.org/aspectj/

Aspect Java started with the source code weaving in 2001, then got the *bytecode weaver* at build time, and use the bytecode weaver at class loading since 2005.

The AOP is a popular and developing technique. Detailed descriptions and examples can be found elsewhere[5].

5 Implementation

We selected Java for portability, and Eclipse[6] with AspectJ plug-in as our implementation tool.

Suppose we have an engine that performs some symbolic calculations. The engine gets data for a computation session as a text file or several files. The engine does not interact with the user during calculation; it is, therefore, a true "black box" that takes data and produces results. We want to wrap the engine in a GUI that collects data, creates text files, runs the engine over these files, and shows results.

A GUI consists of the constant part and the variable part. The constant part contains the session management: storing data for each session, their modification, etc. We also found useful a notion of *environment*, or partially defined session [1]. Each session can be based on an environment where some data are already defined. The environment management is implemented like the session management.

Other features of the constant part of a GUI are possibilities to select one of several engines, to start external programs, to check collected data, to show help, etc.

Modules that enter the data form the variable part of a GUI. These modules depend on the problems solved by a particular engine.

During the assembly of a GUI its constant part is taken as the base. The developer prepares list of data and defines how they have to be entered in the GUI (by selection from several variants, by marking, by text editing, by 2D formula input, by entering parameters of a mathematical object using a wizard, etc.) Each possible method of

[5] http://en.wikipedia.org/wiki/Aspect_oriented_programming
[6] http://www.eclipse.org/

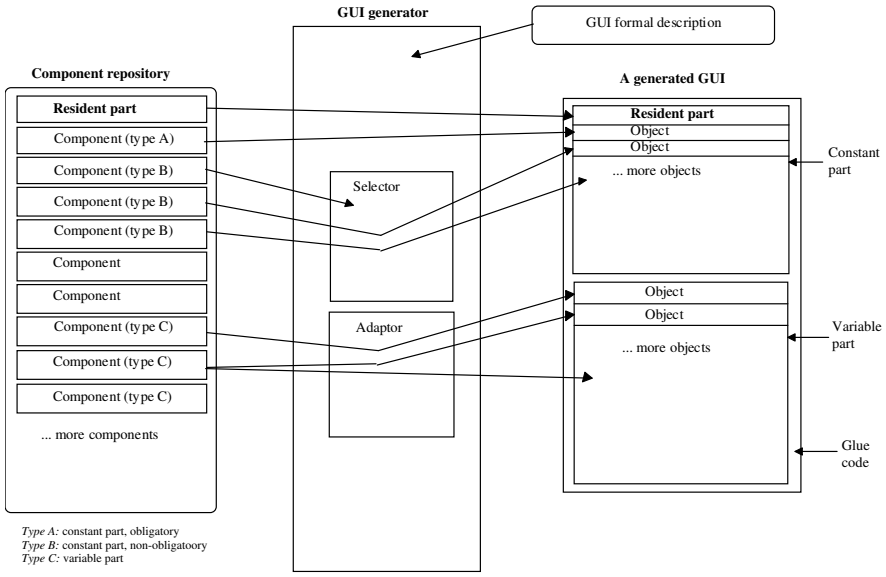

Figure 2. Generation of GUI in our project

the data input is implemented as a customizable component. The necessary modules pass the customization and are glued together with the constant part of the GUI.

The system consists therefore of a pre-implemented constant part, a set of data input components, and a GUI generator that adapts and assembles all parts together.

We already noted that the generated GUI contains the constant and the variable part. The constant part performs a lot of independent standard tasks (e.g., "Save session as..." or "Select engine"). We apply the CP techniques at the development of both parts of GUI. The constant part is also composed from several components and the *resident part*. There is a possibility to vary the constant part. Some components of the constant part are not obligatory. The GUI developer marks included components of the constant part in the list (default is "Include all").

There is some difference between components of the constant and variable GUI parts. The former do not need adaptation; the question is "to use or not to use". The latter are customizable through parametrization.

The generation of a GUI can be illustrated by Fig. 2.

The resident part contains, in particular, the *control center*. The control center registers all assembled components, collects and keeps data from data input components, and produces files for the engine.

The following techniques can be used to assemble applications from components:

- Manual assembly; the gluing code is written manually.
- Visual assembly in an IDE (Java Builder designer, etc.).
- Automated assembly.

The first two techniques are not suitable because they are oriented mainly towards a professional software developer. We selected semi-automated script-based assembly. At first it is necessary to plan the menu structure. The planned menu structure is fixed in an XML description. Using the XML description, a Java menu source is generated programmatically. For each menu item, its action is generated as an aspect. The menu, the aspects and the constant part of the GUI are weaved together resulting in a ready-made GUI.

Fig. 3 shows a simplified GUI that was generated using this technique. The opened dialog permits 2D input of polynomials. To assemble this component to the GUI, it was necessary to generate 2 lines of code in the Java menu source, and an aspect source of 11 lines (5 significant lines), in total 13(7) lines. The constant part of the GUI and the component sources were not changed at all.

6 Requirements to GUI components

Lüer and Rosenblum [10] define a component as a unit of independent deployment prepared for reuse that does not have persistent state. The

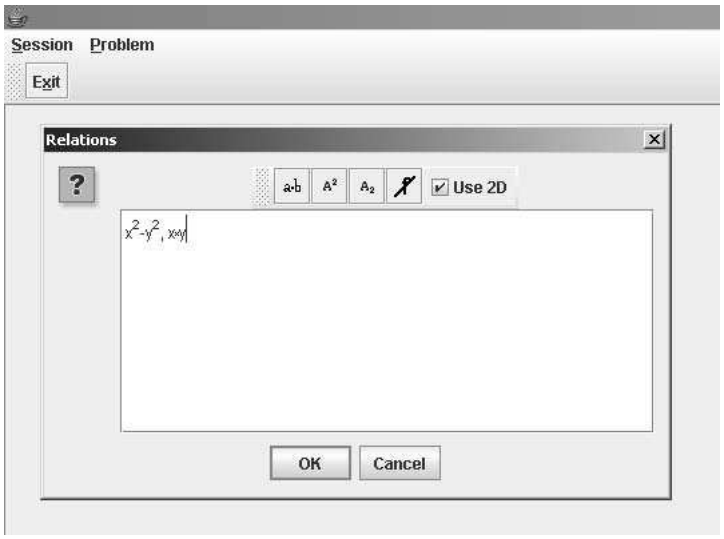

Figure 3. 2D input component in a generated GUI

latter means that a component is a set of classes and does not contain objects.

The following seven requirements of component development were formulated in [10].

1. We should follow the common principles of modular design. In particular, the private and public parts of a component should be separated.

2. Component should be self-descriptive.

3. Component interfaces should be accessed within a global name space through unique names.

4. Development process has two parts, component development and application composition.

5. Application development consists of the component composition and the implementation of additional functionality that is not available in components.

6. We should provide multiple views: a component developer view, a composition view, a type view, an instance view, an overall structure view, etc.

7. The best component reuse is achieved through reuse by reference.

Now we discuss the application of these seven requirements in our case.

1. **Modular design.** The public part of a component should contain public interface definitions, and, optionally, installation tools and self-description. The private part contains implementation (set of classes) and resources (e.g., graphic or help files).

   In [10], installation tools and self-description are exposed as obligatory subparts of component's public part. Installation means providing instances of the classes implemented into the component (objects) for component's clients. It is straightforward in the simplest case; however, a more complicated component may internally select from different implementations of the same published interface.

   In our case, separation of public and private parts will be strictly adhered. Installation means for us parametrization and inclusion in the generated GUI, i.e., gluing. This is made using aspects as described above in Sec. 5.

2. **Self-description.** It is used in a limited way in industrial component models like Java Beans, COM, and .NET. Lüer and Rosenblum [10, p. 4] define five levels of self-description:

   1. The *syntactic* level. It provides signatures of abstract data types that are provided by the component or required by it. A special language (IDL) can be used for such descriptions. We saw that this is a common feature in many CP platforms.

2. The *behavioral* level. It provides semantic description of data types. The semantic description can be informal, semi-formal, or even formal.
3. The *synchronization* level. It provides information that permits cooperating components to resolve concurrency problems.
4. The *QoS,* or *non-functional,* level. QoS (*Quality of Service*) is the usual term for all non-functional specifications like the volume of used memory, the response delay, the result precision, etc.
5. The *non-technical* level. It provides other information like price, contact address, support phone, gained official certifications, etc.

Self-description is less necessary as our components will be used by the GUI generator in programmed manner. The corresponding parts of aspects are in fact self-description in the syntactic level.

3. **Global naming scheme.** We will use our components inside a monolithic system; therefore, we have to use unique names.

4. **Two-stage development process.** In our case, component development and component usage are strictly separated in time. The assembly of a GUI is executed by a different developer, probably by a mathematician that programmed a calculation engine and wants the GUI for it.

5. **Functionality not present in components.** The constant part of GUI provides such functionality in our case. Components are dialogs or wizards to enter one or more session parameters.

6. **Multiple views.** In our case, the component development and their composition are strictly separated in time. The instance view may be useful during the GUI generation.

7. **Reuse by reference.** This principle is applicable in the case when the components have different sources and need maintenance, e.g., distant update. Such option is not supposed in our development, but we could provide it later. If the resulting system became widely used, the existence of a central component repository may be useful.

7 Conclusions

There exists many classifications of component assembly paradigms. One such classification was proposed by a research group[7] at the University of Waterloo, Canada. The classification is as follows:

Design Patterns describe how to assemble objects basing on the separation of concerns. Design patterns capture design experience "in-the-small". The viewpoint approach to software design helps the designer to isolate many design pattern constructs.

Frameworks are semi-finished software architectures for a specific application domain, and they attempt to capture design experience "in-the-large". Frameworks represent the highest level of reusability currently found in software design and implementation. Design patterns and frameworks are related, in that design patterns are normally used to assemble components in a framework.

We have a semi-finished GUI (constant part) and a set of components. Therefore our solution can be classified as *a framework* that is based on the AOP and automates the CP. This is a successful attempt to widen the area of the AOP towards the realm of components, at least for adaptable monolithic applications.

[7] http://csg.uwaterloo.ca/program.html

Acknowledgements

The work was supported by the INTAS grant Ref. Nr. 05–104–7553 "Interface generating toolkit for symbolic computation systems". 2D input component was adapted by dr. Ludmila Burtseva.

References

[1] S. Cojocaru, L. Malahova, A. Colesnicov. *Interfaces to symbolic computation systems: reconsidering experience of Bergman.* // Computer Science Journal of Moldova, vol. **13**, no. 2(28), 2005, p. 232–244.

[2] K. Lieberherr, D. Lorenz, M. Mezini. *Programming with Aspectual Components.* College of Computer Science, Northeastern University, Boston, MA, Report #NU–CCS–99–01, March, 1999.

[3] I.-G. Kim, D.-H. Bae. *Dimensions of Composition Models for Supporting Software System Evolution.* Technical report CS/TR–2005–244, Dept. of Electrical Engineering and Computer Science, KAIST, Daejon, Korea, December 7, 2005.

[4] A.J.A. Wang, K. Qian. *Component-oriented programming*, A Wiley-Interscience publication, 2005, ISBN 0–471–64446–3.

[5] M. Büchi, W. Weck. *A Plea for GreyBox Components.* Turku Centre for Computer Science, TUCS Technical Report No. 122, August 1997, ISBN 952-12-0047-2, ISSN 1239-1891.

[6] J. Bosch. *Composition through Superimposition.* / In: Object-Oriented Technology – ECOOP'97 Workshop Reader, J. Bosch, S. Mitchell (eds.), LNCS 1357, Springer-Verlag, 1997.

[7] G. Heineman. *A model for designing adaptable software components.* / Proceedings, 22nd International Conference on Computer Software and Applications Conference (COMPSAC), Vienna, Austria, Aug. 1998, p. 121–127.

[8] R. Keller, U. Holzle. *Binary Component Adaptation.* Department of Computer Science, University of California, Santa Barbara, CA 93106, Technical Report TRCS97-20, December 3, 1997.

[9] A. Colesnicov, L. Malahona. *Some interface problems between a compiler and a conversational debugger.* Buletinul Academiei de Ştiinţe a Republicii Moldova, Matematica, Nr. 3(6), 1991, p. 12–20, ISSN 0236–3089. – In Russian.

[10] C. Lüer, D. Rosenblum. *WREN – An Environment for Component-Based Development.* Department of Information and Computer Science, University of California, Irvine, CA 92697, Technical Report #00–28, September 1, 2000.

Received April 10, 2007

A. Colesnicov, L. Malahova,

Institute of Mathematics and Computer Science,
5 Academiei str.
Chişinău, MD–2028, Moldova.
E–mail: *kae@math.md*, *mal@math.md*