Essentials of ComponentWare

Wolfgang Pree
Applied Computer Science
University of Constance
D-78457 Constance, Germany
Fon: +49.7531.88.44.33
Fax: +49.7531.88.35.77
pree@acm.org

Abstract. Component-based software development is associated with a shift from statement-oriented coding to system building by plugging together components. The idea is not new and some progress has been made over the past decades. Nevertheless, today's software development practice is still far away from the vision. In recent years, the term componentware became the vogue in the software engineering community. The paper attempts to define the relevant terms by relating the definitions to the already well-defined terms of object technology. In particular, the focus lies on a discussion of the deficiencies of the object-oriented paradigm and how componentware might solve these short comings. Finally, the paper tries to answer the question whether some enhancements of the object-oriented paradigm indeed represent the dawn of a new era of software development.

Keywords. Component-based software development, object-oriented programming, frameworks, dynamic computing, visual programming.

1 Introduction

Though object technology has become the vogue in the software engineering community, quite a lot of projects regarded as being object-oriented failed in recent years. Of course, the term failure has different meanings depending on various circumstances. Nevertheless, it expresses at least that object technology has not met the expectations in those projects. Both organizational and technical troubles might have caused these failures. Primarily, adopters of the object-oriented paradigm graduate to this camp in order to significantly improve reusability and the overall quality of software systems.

The buzzword *componentware* is now heralded as the next wave to fulfill the promises that object technology could not deliver. In addition, proponents of component-based software development view this technology as means to let end user computing become reality. This paper contributes to a clarification of terms by discussing the overlaping concepts underlying object and component technology and the few additional ingredients that might indeed justify the creation of the new term componentware.

Central to the component paradigm are components and their interaction. So far several ways of specifiying components have been proposed. We disagree with the point of view of authors who regard numerous language concepts as foundation of components. For example, Nierstrasz and Dami (1995, p.4) write: “Mixins, functions, macros, procedures, templates and modules may all be valid examples of components”. Lakos (1996, p.32) describes components as follows:

"Conceptually a component embodies a subset of the logical design that makes sense to exist as an independent, cohesive unit. Classes, functions, enumeration and so on are the logical entities that make up these components". In order to come up with a more concise definition, we first take a look at what should be improved in the object-oriented paradigm.

Deficiencies of the object-oriented paradigm

Figure 1 illustrates the problems associated with the object-oriented paradigm. Classes/objects implemented in one programming language cannot interoperate with those implemented in other languages. In some object-oriented languages even the same compiler version has to be used so that objects become interoperable. Furthermore, composition of objects is typically done on the language level. Black-box composition support is missing, that is, visual/interactive tools that allow the plugging together of objects.

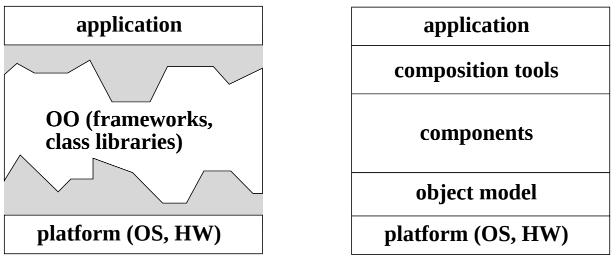

Figure 1 Object-orientation versus componentware (adapted from Weinand, 1996).

2 Component-based software development

This section sketches current approaches of implementing systems with components. We start with a description of what components essentially are and how components and their interactions are formulated in programming languages and/or interactive visual environments.

2.1 Components as Abstract Data Types (ADTs)

A component is simply a data capsule. Thus information hiding becomes the core construction principle underlying components. Parnas (1972, p.1056) defines information hiding as follows: "A module is characterized by its knowledge of a design decision which it hides from all others. Its interface was chosen to reveal as little as possible about its inner working." Several ways of bringing information hiding down to earth have been proposed:

Components in module-oriented languages

In module-oriented languages such as Modula-2 and Ada components are called modules. A module offers an interface to its clients. Clients interact with a module only through its interface. The internal implementation in the module's body is hidden from them. Though each module defined in Ada and Modula-2 represents an abstract data structure (ADS), module-oriented languages also allow the definition of ADTs.

Components expressed in module-oriented languages can only be reused in another project exactly in the same way as the module was originally designed. Otherwise, the source code or even its interface have to be changed. This leads to the creation of multiple versions. Component testing would have to be done again. In other words, such reuse is far from the ideal world. Unfortunately, most modules require slight changes to be reusable in other software systems. Only quite basic modules, such as data structures and GUI dialog items, allow black-box reuse (reuse without any modifications).

Components in object-oriented languages

In object-oriented languages such as Smalltalk, C++ and Java, components are instances of classes. A class represents an abstract data type (ADT). Analogous to modules, a class offers an interface and hides its realization. In contrast to a module, a class serves as a component factory by allowing the instantiation of any number of objects.

In order to overcome some reuse problems of module-oriented languages, object-oriented languages introduce language constructs to achieve *delta changes* (programming by difference) without having to touch the source code of original modules/classes. Inheritance is central to this solution: A subclass defines the delta by which a class differs from its superclass. In other words, inheritance allows ADT adaptations without having to edit source code or give up compatibility. To sum up, object-oriented languages improve the module concept. They allow a straightforward definition of ADTs and provide language constructs for their extension and modification.

As a consequence, many adopters of object technology expect that the usage of an object-oriented language alone yields significant improvements in software reusability. This leads to a quite naive application of object technology as corroborated in Section 3.

Language-independent component specification

The goal of componet-based software development is that a component can be implemented in (almost) any language, not only in any module-oriented and object-oriented languages but even in conventional languages. As a consequence, standards how to describe componets have been established. The component (module) interface is described either

- textually by means of an interface description language (IDL)
- visually/interactively by appropriate tools.

In case of using OO languages, several classes (fine-grained components) typically form one coarse-grained component (see Figure 2). The reason for this is that language-independent component models offer only a common denominator of features for defining component interfaces. Thus they allow only the definition of coarse-grained components that correspond to subsystems rather than to single classes.

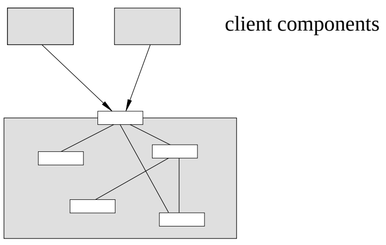

Figure 2 Granularity of components.

Conventional, procedural languages lack the language features necessary to define ADTs. Interactive, visual component programming environments on top of procedural languages offer both, the additional language features for defining components and means for connecting them. Interactive, visual component programming environments on top of object-oriented or module-oriented languages essentially allow the specification of object interactions.

For example, Visual Basic augments the underlying procedural language so that components can be expressed. The resulting components, now called ActiveXs, differ slightly from modules in module-oriented languages: components may provide any number of interfaces. Parts (ParcPlace-Digitalk, 1997) and VisualAge (IBM, 1997) are based on Smalltalk and represent further examples of interactive environments for defining the interaction between objects.

2.2 Interoperability standards

Component standards such as (D)SOM (Component Integration Labs, 1996) and (D)COM (Brockschmidt, 1995) represent one approach to ameliorate the problem of making components interoperable. The socalled *object models* SOM (System Object Model) and COM (Component Object Model) indeed make different languages and compilers interoperable. Unfortunately, these standards have an unnecessary inherent complexity and fail to offer meta-information and proper garbage collection, which is simply a must for extensible systems. Java Beans (Sun, 1997) represents a new component model that offers also bridges to the object models (D)SOM and (D)COM. Time will show which of the object/component models will become the predominant standard.

OpenDoc (based on SOM; its further development was canceled by Apple and IBM in early 1997) and OLE (Object Linking and Embedding; based on COM) are first framework architectures that build on these object models and manage resources (such as the keyboard and the screen) between interoperating components. Figure 3 illustrates schematically the relationship between the component models and compound document frameworks.

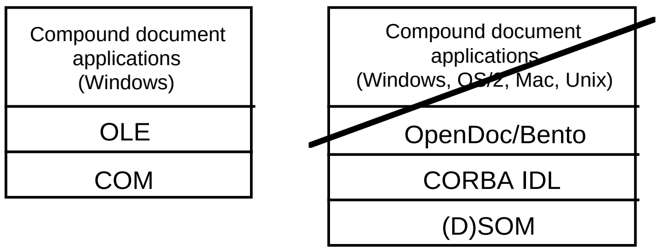
Compound document applications (Windows)
OLE
COM
Compound document applications (Windows, OS/2, Mac, Unix)
OpenDoc/Bento
CORBA IDL
(D)SOM

Figure 3 Component models and compound document architectures.

3 Dynamic computing perspective

Many software systems belong to the category of fatware (Wirth, 1995). For example, take a look at typical desktop application:

- Spreadsheets have more than 1000 features
- The applications require numerous MBs on hard disks and in RAM

Instead, users should work with a lean application that offers the typical features needed. Additional components are only loaded when they are required. Language systems must support the flexible integration of components into running software, independent of the hardware and operating platform where the particular software executes. For this purpose, the language system has to offer appropriate meta-information and dynamic linking of components.

Java (Sun, 1997) and the Gadgets (Gutknecht, 1994) derivate of the Oberon system (Wirth and Gutknecht, 1992) exemplify how the portability problem can be solved: machine-independent code is dynamically linked as a component is loaded. The component model represented by Java Beans (Sun, 1996) will further contribute to platform-independent components. Marimba’s Castanet (Marimba, 1997) for automatic, incremental updates of single application components supports dynamic computing in that components are only loaded when they are not up-to-date any longer.

Envisioned end user programming systems

Dynamic computing implies distributing software components that can be plugged into software systems. The underlying technology will be frameworks whose behavior is modified and/or extended by composition. The Internet could strongly boost such a component market. Pree (1995) and Goldberg (1995), for example, provide a thorough discussion regarding framework technology.

Currently software components are rather monolithic. In many cases components materialize as either simple GUI items or full-fledged applications. Much more levels of granularity of software components can be expected. Visual, interactive composition tools will become available that allow end users to configure software systems by handling components that specialize framework hot spots.

Ted Lewis (1996, p.84) corroborates our point of view that “we have the technology to solve most of the problems left unresolved by the software engineering elite”. For example, a technically feasible response to the problem that many projects start from scratch would be parts assembly tools replacing programming languages. Small-scale, pluggable components represent an alternative to large-scale, monolithic systems. Ted Lewis forsees a software economy driven by the demand to solve these and other decades-long unresolved issues and the feasability of meeting that demand. One consequence will be the rise of a “cottage industry of application-specific framework developers—a small corps of elite craftspersons” (Lewis, 1996, p.84).

Overall, frameworks will remain the long-term players towards reaching the goal of developing software with a building-block approach. Nevertheless, the creation of frameworks and their corresponding specializing components will clearly be separated from their consumption. Ideally, people who are not familiar with current programming languages will configure their domain-specific systems. Component-based software development could indeed form a new paradigm software engineering.

References

Boehm B. (1994): *Megaprogramming*. Video tape by University Video Communications (http://www.uvc.com), Stanford, California

Brockschmidt K. (1995): *Inside OLE.* Redmont, Washington, Microsoft Press

Goldberg A., Rubin K. (1995): *Suceeding with Objects: Decision Frameworks for Project Management*. Reading, Massachusetts: Addison-Wesley

Gutknecht J. (1994): *Oberon System 3: Vision of a Future Software Technology*. Software Concepts & Tools, 15(1), 26-33

IBM (1997): *VisualAge for Smalltalk, User`s Guide*. IBM

Lakos J. (1996): *Large Scale C++ Software Design*. C++ Report, 8(6), 27-37

Lewis (1996): *The Next 10.000$_2$ Years: Part II*. IEEE Computer, May, 78-86

Marimba (1997): *Castanet description & demonstration*. White Papers at http://www.marimba.com, Marimba Inc.

Nierstrasz O., Dami L. (1995): *Component-Oriented Software Technology*. In: Object-Oriented Software Composition, Nierstrasz O, Tsichtitzis D, Prentice Hall, 3-28

ParcPlace-Digitalk (1997): *Parts Workbench User’s Guide*. ParcPlace-Digitalk Inc.

Parnas D.L. (1972): *On the Criteria to be Used in Decomposing Systems into Modules*. *Communications of the ACM*, 15(12), 1053-1058

Pree W. (1995): *Design Patterns for Object-Oriented Software Development*. Reading, Massachusetts: Addison-Wesley

Sun (1997): *The Java Language*; *Java Beans*. White Papers at http://java.sun.com, Sun Microsystems

Weinand A. (1996): *Komponentenbasierte Softwareentwicklung*. Tutorial (in German), OOP'96 Conference, Munich

Wirth N., Gutknecht J. (1992): *Project Oberon.* Wokingham. Addison-Wesley/ACM Press

Wirth N. (1995): *A Plea for Lean Software,* IEEE Computer, 28(2)