Software—Concepts and Tools (1997) 18: 169–174

Software—Concepts and Tools
© Springer-Verlag 1997

Component-Based Software Development—A New Paradigm in Software Engineering?

Wolfgang Pree
Software Engineering Group, University of Constance, D-78457 Constance, Germany
e-mail: 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 has become the vogue in the software engineering community. This paper defines the relevant terms by relating them to the already well-established 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 shortcomings. Furthermore, the role of object-oriented frameworks as underlying technology of plug & play software is illustrated. Finally, the paper tries to answer the question of 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 have 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 migrate 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 a means to let end-user computing become reality. This paper contributes to a clarification of terms by discussing the overlapping 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. Several ways of specifying components have been proposed. We disagree with the point of view of authors who regard numerous language concepts as foundations of components. For example, Nierstrasz and Dami [10] write "Mixins, functions, macros, procedures, templates and modules may all be valid examples of components". Lakos [6] 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, enumerations and so on are the logical entities that make up these components". In order to formulate a more useful 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. Some object-oriented languages even require the same compiler version. Furthermore, composition of objects is typically done on the language level. Composition support is missing, that is, visual/ interactive tools that allow the plugging together of objects.

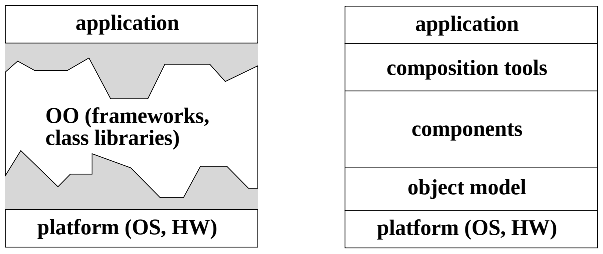

Figure 1. Object-orientation versus componentware (adapted from Weinand [16]).

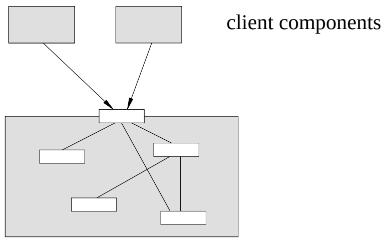

Figure 2. Granularity of components.

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

A component is simply a data capsule. Thus information hiding becomes the core construction principle underlying components. Parnas [12] 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 clients. Though each module defined in Ada and Modula-2 represents an abstract data structure (ADS), module-oriented languages also allow the definition of abstract data types (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 must be repeated. 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 (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 the usage of an object-oriented language alone to yield 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 component-based software development is the possibility to implement a component in (almost) any language, not only in any module-oriented and object-oriented languages but even in conventional languages. As a consequence, standards for describing components have been established. The component (module) interface is described either

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

In OO languages, several classes (fine-grained components) typically form one coarse-grained

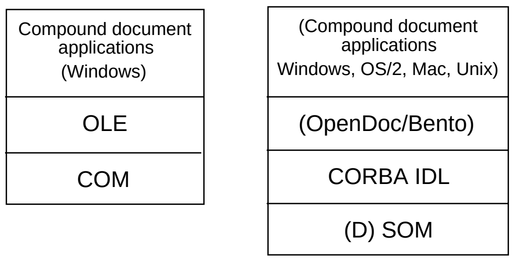
Compound document applications (Windows)
OLE
COM
(Compound document applications Windows, OS/2, Mac, Unix)
(OpenDoc/Bento)
CORBA IDL
(D) SOM

Figure 3. Component models and compound document architectures.

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 rather to subsystems than to single classes.

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 a 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 to allow expression of components. The resulting components, now called ActiveXs, differ slightly from modules. ActiveX components may provide any number of interfaces. Parts [11] and VisualAge [5] 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 and (D)COM [2] represent one approach to ameliorate the problem of making components interoperable. The *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 [14] represents a new component model that also offers 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)

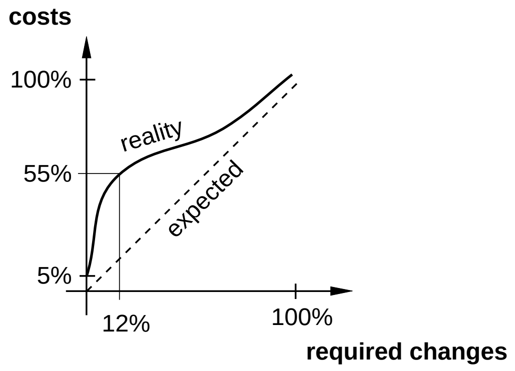

Figure 4. Costs of component changes versus costs of development from scratch (adapted from Boehm [1]).

are 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.

3. Frameworks as Core Technology of Plug & Play Software

In general, reuse of software has obvious advantages, such as reduced development and maintenance costs and a positive impact on software quality (already tested components contribute to correct software). Components based on efficient algorithms allow the construction of more efficient software. Reuse of portable software components implies a more portable system. But how can software reuse become reality?

Though organizational measures form a crucial precondition to successful reuse, we focus on technical and economic issues and discuss the implications of single component reuse as opposed to framework reuse. In general, more flexibility implies more programming effort by the programmers who reuse the software components. The ordering below reflects decreasing levels of flexibility and thus increasing ease of reuse. Black-box frameworks represent the most rigid albeit the easiest way of reusing components; thus they form the backbone of pure componentware.

3.1 Naive View: Reuse Based on Lego Kit Component Libraries

Single component reuse means that programmers build the overall software system architecture on their own. They have to locate the appropriate components in a Lego kit library and define their interaction.

Barry Boehm (1994) presents a study regarding the reuse costs of single components. The study was conducted in NASA's software engineering lab and comprises a sample of about 3000 modules. Figure 4 summarizes the results; the graph relates the percentage of necessary changes to a single component in order to render it reusable (x-axis) to the costs of these changes relative to the development of a component from scratch (y-axis).

First, reuse of single components without any changes does not come for free. One has to locate the appropriate component, understand its interface and determine how to fit it into the system under construction. Furthermore, a (hypertext-based) catalog of reusable components and the components themselves have to be maintained. The components themselves have to be built for reuse, which is more expensive than for a special purpose. More intimidating is the fact that only a few changes (12%) to a component raise the reuse costs to 55% compared to a from-scratch development of the particular component; not the changes themselves but the required component understanding costs a lot. Without this understanding, changes could cause disastrous side effects.

The study corroborates programmer's gut feeling that single-component reuse is almost as expensive as development from scratch. Though one might argue that a 45% cost reduction is a significant improvement compared to component development from scratch, the following psycological aspect has to be considered, too. Programmers often do not trust software written by others (*not-invented-here* syndrome). Furthermore, the code to plug together several single components has to be written. Assuming that 50% of the resulting overall system consists of reused single components, the overall cost reduction is only 22.5% (45%/2) or even less. As how single components should be connected might not be completely clear, a significant amount of bulky extra code could be the price of plugging together single components.

The bottom line is that it does not matter whether languages support adaptations in an elegant way or not. Simply using an object-oriented language instead of a conventional one or using object/component models cannot ensure reusability improvements. Projects that apply components with this naive view are likely to fail.

3.2 Framework Reuse

Instead of reusing single components, most successful component-based projects practice framework development and reuse. A framework is simply a collection of several single components with predefined cooperations between them, for the purpose of accomplishing a certain task. Some of these single components are designed to be replaceable, typically corresponding to abstract classes in the framework's class hierarchy. We call the points of predefined refinement hot spots [13]. Figure 5 shows these framework characteristics with the hot spots in gray.

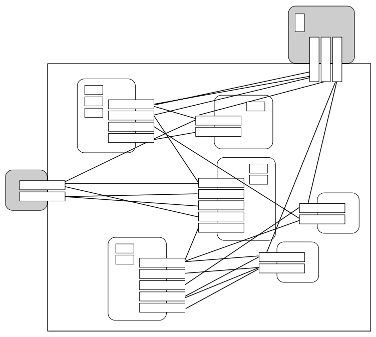

Figure 5. Framework with hot spots.

A framework deserves the attribute well-designed if it offers the domain-specific hot spots to achieve the desired flexibility via adaptation of these hot spots. Well-designed frameworks also predefine most of the overall architecture, i.e., the composition and interaction of its components. The lines connecting method interfaces in Figure 5 express this glue between the components. Applications built on top of a framework reuse not only source code but also architecture design, which we consider as the most important characteristic of frameworks.

Note that the framework concept is quite independent of how components are implemented. Frameworks just require that components can be replaced by more specific ones that are compatible to the original placeholders. Of course, object-oriented languages support specialization in a straightforward manner through inheritance.

The bad news is that frameworks require enormous development effort. Many problems result from complicated interaction scenarios between partially defined components. The costs of developing a framework are significantly higher compared to the development costs of a specific application. So frameworks represent a long-term investment that pays off only if similar applications are developed again and again in a domain.

Furthermore, tools and methods assisting in framework development are almost non-existent or in their infancy. Framework technology itself is not yet mature. For example, it remains unclear how frameworks designed by different teams with separate control flows can interoperate. The fragile base-class problem [8] might overturn fundamental framework design decisions. Changes in base classes of a framework can cascade onto numerous classes inheriting from them.

Finally, framework development and reuse conflicts with the current project culture that tries to optimize the development of specific software solutions instead of generic ones. As this paper focuses on technical aspects, we refer to the excellent discussion of organizational issues by Goldberg and Rubin [3].

Despite the mentioned problems with the state of the art in framework technology, (black-box) frameworks provide the enabling technology of plug & play software, where most adaptations can be achieved by exchanging components.

4. Dynamic Computing Perspective

Many software systems belong to the category of fatware [18]. For example, a typical desktop application such as a spreadsheet has 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 runs. For this purpose, the language system has to offer appropriate meta-information and dynamic linking of components.

Java [14], ET++ [15] and the Gadgets [4] derivative of the Oberon system [17] 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 [14] will further contribute to platform-independent components. Marimba's Castanet [9] for automatic, incremental updates of single application components supports dynamic computing in that components are only loaded when they are no longer up-to-date.

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.

Currently software components are rather monolithic. In many cases components materialize as either simple GUI items or full-fledged applications. Many 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 [7] 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, plugable components represent an alternative to large-scale, monolithic systems. Ted Lewis foresees a software economy driven by the demand to solve these and other decades-long unresolved issues and the feasibility 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" [7].

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, domain experts who are not familiar with current programming languages will configure their domain-specific systems. Thus, component-based software development could indeed represent a new paradigm in software engineering.

References

1. Boehm B (1994) Megaprogramming. Video tape by University Video Communications, Stanford, California, (http://www.uvc.com)
2. Brockschmidt K (1995) Inside OLE. Redmont, Washington, Microsoft Press
3. Goldberg A, Rubin K (1995) Succeeding with Objects: Decision Frameworks for Project Management. Reading, Massachusetts Addison-Wesley
4. Gutknecht J (1994) Oberon System 3: Vision of a Future Software Technology. Software Concepts & Tools, 15(1), 26-33
5. IBM (1997) VisualAge for Smalltalk, User's Guide. IBM
6. Lakos J (1996) Large Scale C++ Software Design. C++ Report, 8(6), 27-37
7. Lewis (1996) The Next 10.000^2 Years Part II. IEEE Computer, May, 78-86
8. Lewis T, et al. (1995) Object-Oriented Application Frameworks. Manning Publications/Prentice Hall
9. Marimba (1997) Castanet description & demonstration. White Papers at Marimba Inc., http://www.marimba.com,
10. Nierstrasz O, Dami L (1995) Component-Oriented Software Technology. In Object-Oriented Software Composition, Nierstrasz O, Tsichtitzis D, Prentice Hall, 3-28
11. ParcPlace-Digitalk (1997) Parts Workbench User's Guide. ParcPlace-Digitalk Inc.
12. Parnas DL (1972) On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 15(12), 1053-1058
13. Pree W (1995) Design Patterns for Object-Oriented Software Development. Reading, Massachusetts Addison-Wesley
14. Sun (1997) The Java Language; Java Beans. White

Papers at http://java.sun.com, Sun Microsystems

15. Weinand A, Gamma E, Marty R (1989) Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. Structured Programming, 10(2), Springer Verlag
16. Weinand A (1996) Komponentenbasierte Software-entwicklung. Tutorial (in German), OOP'96 Conference, Munich
17. Wirth N, Gutknecht J (1992) Project Oberon. Wokingham. Addison-Wesley/ACM Press
18. Wirth N (1995) A Plea for Lean Software, IEEE Computer, 28(2)