Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 56 (2005) 41–57
Science of Computer Programming
www.elsevier.com/locate/scico

Supporting software composition at the programming language level

Peter H. Fröhlich[a,*], Andreas Gal[b], Michael Franz[b]

[a] *Department of Computer Science and Engineering, University of California, Riverside, CA, United States*
[b] *School of Information and Computer Science, University of California, Irvine, CA, United States*

Received 31 October 2003; received in revised form 22 August 2004; accepted 6 September 2004
Available online 9 December 2004

Abstract

We are in the midst of a paradigm shift toward component-oriented software development, and significant progress has been made in understanding and harnessing this new paradigm. Oddly enough, however, the new paradigm does not currently extend to the level at which components themselves are constructed. While we have composition architectures and languages that describe how systems are put together out of atomic program parts, the parts themselves are still constructed on the basis of a previous paradigm: object-oriented programming. We argue that this mismatch impedes the progress of compositional software design: many of the assumptions that underlie object-oriented languages simply do not apply in the open and dynamic contexts of component software environments. What, then, would a programming language that supported component-oriented programming at the smallest granularity look like? Lag*oo*na, our project to develop such a language, tries to answer this question. This paper motivates the key concepts behind Lag*oo*na and briefly describes their realization (using Lag*oo*na itself as the implementation language) in the context of Microsoft's .NET environment.

Keywords: Component-oriented software development; Programming languages; Distributed extensibility; Language paradigms beyond object-oriented programming

* Corresponding author. Tel.: +1 951 827 2604; fax: +1 951 827 4643.
E-mail address: phf@acm.org (P.H. Fröhlich).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.11.004

1. Introduction

While the idea of "software components" was proposed as far back as the 1960s [23], the arrival of the Internet has propelled us into an age where component-oriented programming (COP) is becoming simultaneously viable and necessary: viable, because an efficient component discovery and distribution mechanism is now available; necessary, because the complexity of Internet-enabled applications often exceeds the abstraction capabilities of existing programming paradigms. There is, of course, much confusion about what COP *actually* means [36,20]. This state of affairs is similar to the confusion surrounding object-oriented programming (OOP) in the 1980s. As with OOP, the "essence" of COP is not primarily found in technical details of programming language design and implementation (although we focus on its *implications* for these areas in the following). Instead, and again similar to the OOP case, the importance of the paradigm lies in its conceptual vision, i.e. the software architectures that it strives for, the software qualities that it emphasizes, and the software development processes that it mandates. The latter is also the most striking difference between COP and established paradigms, which are usually silent on issues of process.

We contend that the "essence" of COP is the notion of *distributed extensibility*, in contrast to the notion of *centralized reuse* which has been the focus of software components since they were first proposed. Centralized reuse means that software components are acquired by an application vendor who in turn sells a monolithic application to users. The application vendor alone has *complete* control over the integration process, deciding which components are delivered as part of the final application. Once deployed, the application cannot be "reintegrated" with newer or different components, keeping the application vendor in control. Also, without "privileged" access to the internals of the application, no party except the application vendor can develop extensions. In contrast, distributed extensibility (see Fig. 1) means that *any* interested party can develop extensions, which can be acquired and integrated by *anyone* at *any* time [12]. Monolithic "applications" disappear under distributed extensibility, to be replaced by *components* and *frameworks* (see Fig. 2). Components provide functional extensions for (domain-specific) frameworks, while frameworks provide (customized) execution environments for components.

The fundamental difference in process between centralized reuse and distributed extensibility also has profound implications for programming languages. It is current practice to *approximate* certain COP ideas using a variety of essentially OOP languages

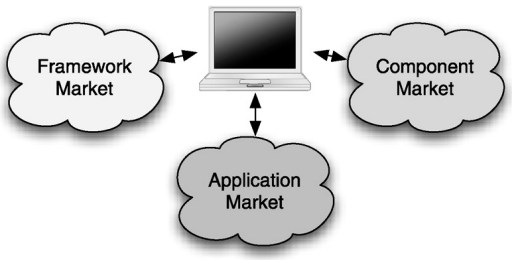

Fig. 1. Distributed extensibility enables anyone to independently develop, acquire, and integrate anything, anytime.