Towards Context-Sensitive Intelligence

Holger Mügge[1], Tobias Rho[1], Marcel Winandy[1], Markus Won[1], Armin B. Cremers[1], Pascal Costanza[2], and Roman Englert[3]

[1] Institute of Computer Science III, University of Bonn, Römerstr. 164, 53117 Bonn, Germany
{muegge, rho, winandy, won, abc}@iai.uni-bonn.de
[2] Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
pc@p-cos.net
[3] Deutsche Telekom Laboratories, Ernst-Reuter-Platz 7, 10587 Berlin, Germany
Roman.Englert@telekom.de

Abstract. Even modern component architectures do not provide for easily manageable context-sensitive adaptability, a key requirement for ambient intelligence. The reason is that components are too large – providing black boxes with adaptation points only at their boundaries – and to small – lacking good means for expressing concerns beyond the scope of single components – at the same time. We present a framework that makes components more fine-grained so that adaptation points inside of them become accessible, and more coarse-grained so that changes of single components result in the necessary update of structurally constrained dependants. This will lead to higher quality applications that fit better into personalized and context-aware usage scenarios.

1 Introduction

Most of the software sold nowadays are off-the-shelf products designed to meet the requirements of very different types of users. One way to meet these requirements is to design software that is flexible in such a way that it can be used in very different contexts. Thus, look and feel, functionality, and behavior have to be tailorable or even adaptive according to the task that needs to be fulfilled. Especially out of an organizational context most users have to tailor their software on their own. Taken into account that experiences in the use of computer systems in general increase exceedingly, tailorable and end user development applications become interesting topics. Component architectures were basically developed with the idea of higher reusability of parts of software. Furthermore, it is shown that they also build a basis for highly flexible software [1]. In this case the same operations that are used to compose software out of single components now can be applied to existing (component-based) software during runtime. Therefore, the basis for a tailoring language consists basically of three kinds of operations: choosing components, parameterizing them, binding

R. Morrison and F. Oquendo (Eds.): EWSA 2005, LNCS 3527, pp. 231–238, 2005.

them together. In this way very simple operations that can be easily understood by end users enhance the possibilities of tailoring software in a powerful way (cf. [2]).

2 Intelligent Dealing with Complexity

Still, there are several open questions according to how tailoring can be eased for end users. For instance, there is a need for a graphical front end that allows visual tailoring techniques. Here one problem is how invisible components can be presented to the users. In different studies it was shown (cf. [1]) that users are able to tailor their GUIs very easily. Nevertheless, the problem of finding an appropriate visual tailoring environment for both - visible and invisible components - is still unsolved.

A second problem is that tailoring becomes harder when more flexibility is needed. Flexibility in component architectures designed for tailorable applications is reached by a higher degree of decomposition [1]. That means, the more components are needed to design software, the more flexible it can be tailored as there are many fine-grained components that can be parameterized or exchanged.

Our goal is to design a stable basis for highly flexible software systems. Component architectures are appropriate in this case and thus concentrating on the second problem.

There are several approaches which may ease the use of software in different contexts. In our case we believe in a combination of tailorable software (user is in an active role) and adaptive techniques (software does adaptions by itself). This might be helpful to cope with the complexity problem. Combining both techniques means that tailoring activities are followed by automatic adaptions of the system which checks for dependencies within the composition and adjusts it.

Another point is the inspection of contexts: How do contexts look like and how can they influence the software system? The abstraction of different use contexts and their explicit description can reduce complexity of the components as context descriptions influence more the whole composition. If the context changes, users have only to switch the current context description which leads to changed functionality of the whole composition.

Furthermore, one source of complexity is that many applications run distributed and networked. In such systems (client-server, peer-2-peer) tailoring becomes even harder as adaptations on one client or one server might have dependencies on another part of the application which runs on a different machine. In such cases server components have to behave according to different clients. In section 3 we describe three basic techniques which can overcome these problems. After that we show how they can be integrated within one component framework in section 4.