The Journal of Systems and Software 85 (2012) 1801–1817

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Context-oriented programming: A software engineering perspective

Guido Salvaneschi [a,b,*], Carlo Ghezzi [b], Matteo Pradella [b]

[a] TU Darmstadt, Software Technology Group, Hochschulstr. 10, 64289 Darmstadt, Germany
[b] DEEPSE Group, DEI, Politecnico di Milano, Piazza L. Da Vinci, 32, 20133 Milano, Italy

ARTICLE INFO

Article history:
Received 30 April 2011
Received in revised form 8 March 2012
Accepted 11 March 2012
Available online 20 March 2012

Keywords:
Context-oriented programming
Context
Context-awareness

ABSTRACT

The implementation of context-aware systems can be supported through the adoption of techniques at the architectural level such as middlewares or component-oriented architectures. It can also be supported by suitable constructs at the programming language level. Context-oriented programming (COP) is emerging as a novel paradigm for the implementation of this kind of software, in particular in the field of mobile and ubiquitous computing. The COP paradigm tackles the issue of developing context-aware systems at the language-level, introducing ad hoc language abstractions to manage adaptations modularization and their dynamic activation. In this paper we review the state of the art in the field of COP in the perspective of the benefits that this technique can provide to software engineers in the design and implementation of context-aware applications.

1. Introduction

Context-awareness is a primary issue in emerging fields such as ubiquitous and mobile computing. In the design of context-aware systems some challenges must be addressed. First, adaptation to the current context is often an aspect that crosscuts the application logic, so it is often orthogonal to the main modularization direction. It is therefore difficult to organize the codebase in a way that does not compromise maintainability and separation of concerns. Furthermore, dynamic adaptation to the context requires that an application modifies its behavior at runtime. While this is not difficult to obtain in principle even with traditional techniques, organizing dynamic behavioral change in a systematic and effective way requires careful engineering.

Over the years, several approaches have been proposed to support the design and development of context-aware software, at different abstraction levels. These approaches mainly encompass software architectures, component-based design, and middleware (Chan and Chuang, 2003; Bellavista et al., 2003; Gu et al., 2005, 2004; Geihs et al., 2009; Capra et al., 2003). Context-oriented programming (COP) (Hirschfeld et al., 2008) was recently proposed as a complementary approach for supporting dynamic adaptation to context conditions. COP provides language-level abstractions to modularize behavioral adaptation concerns and to dynamically activate them during the program execution.

In this work we review the achievements of context-oriented programming and research advances in the perspective of the benefits they can bring to the software engineering community in general and specifically to the engineering of context-aware systems community. While so far COP has been mainly studied from a programming language point of view, we argue that it can empower software engineers committed to the design of context-aware systems with a very powerful approach and tool. The COP paradigm provides an additional dimension to standard programming techniques to dynamically switch among the behaviors associated with each context, such as bandwidth availability, presence of WiFi or data connection, battery level, or current system workload. In addition, COP provides means to dynamically combine different behaviors when all the associated contexts are active at the same time and properly modularize the code for each behavior.

Our work tries to bridge the gap between the programming languages and the software engineering communities, covering the fundamental aspects which can be of interest for a software architect such as the compilation process, modularization, dynamic activation of adaptations, and consistency of behavioral variations.

Starting from the pioneering work of Costanza and Hirschfeld (2005), COP evolved in a variety of solutions addressing in different ways the problems of behavioral variations modularization and their dynamic activation. The proposed languages share the concept of language-level runtime adaptation to context, but interpret the paradigm in different ways. Therefore, at present COP is constituted by a family of languages specifically developed to support context adaptation, with some widely adopted design solutions and many (sometimes radically) different variants.

In the exposition we adopted the following criteria. We set up our analysis centering it on layer-based (Hirschfeld et al., 2008)

* Corresponding author at: TU Darmstadt, Software Technology Group, Hochschulstr. 10, 64289 Darmstadt, Germany.
E-mail addresses: salvaneschi@elet.polimi.it (G. Salvaneschi), ghezzi@elet.polimi.it (C. Ghezzi), pradella@elet.polimi.it (M. Pradella).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2012.03.024

COP languages, because they represent the majority of the existing approaches and the most influential efforts in the research community. Single implementations which do not fall into this category and present ad hoc solutions are also introduced, if they present a feature that is particularly relevant to the discussion.

The paper is organized as follows: In Section 2 we introduce the fundamental concepts of COP. In Section 3 we present the main COP flavors that have been implemented so far. In Section 4 we show an overview of COP existing software and application areas. In Section 5 we analyze the related work and in Section 6 we discuss a roadmap for future research. Section 7 draws the conclusions.

2. Fundamental concepts

In this section we present the foundations of the COP paradigm. We adopt a top-down approach, which starts from the abstract notion of context and then focuses on behavioral variations that conceptually enable context-aware adaptation.

The notion of *context* traditionally adopted in COP is open and pragmatic: any computationally accessible information can be considered as context (Hirschfeld et al., 2008). Such a definition can be surprisingly vague, but practice with context-aware systems confirms its validity. First of all, the definition does not limit the context to the information reaching the system from *outside* (i.e. the *environment*), but it also encompasses information originating *inside* the system boundaries, such as performance monitoring or intrusion detection. Moreover, such a general approach does not prescribe any restriction to the level of abstraction through which the context is represented inside the system. In fact in a typical context-aware application context information is first obtained by sensors in the form of *numerical* observables. For example, the bandwidth consumption on a network interface can be quantified as 100 Mb/s or the processor is observed to be busy 95% of the time. Often these values are abstracted and combined to obtain some *symbolic* observable. For example, the measured processor usage can be associated with the *heavy load* condition.

A point on which COP approaches differ is whether a unique global context exists or different parts of the system can live in separate contexts. The Ambience programming language (González et al., 2008) adopts a model whereby the whole application shares the same global context. This model reflects the intuitive idea that there is only one real-world context. Conversely, most COP languages (Appeltauer et al., 2009a) do not enforce uniqueness of context and therefore different parts of the application, for example different threads, can live in different contexts and therefore adapt their behavior differently. While from a conceptual point of view a unique context leads to a more elegant and intuitive model, the possibility of exploiting multiple contexts in the same application is more flexible in practice. For example it allows each thread of a server-side software to independently adapt to the specific conditions of the client which is currently in charge of.

A key concept of COP is the *behavioral variation* (Hirschfeld et al., 2008), which is a unit of behavior that can be made effective (partially) modifying the overall behavior of the application. A behavioral variation is enabled by means of a variation *activation*. Runtime context adaptation is achieved in COP by dynamically (i.e. during the execution) activating behavioral variations. When multiple variations are active at the same time, they dynamically *combine* to generate the emerging application behavior. In COP the role of variations is twofold. On the one hand, they allow dynamic activation of a behavioral change; on the other hand, they are the *modularization unit* of such behavioral fragment.

Over the years, this conceptual framework has been interpreted in different ways, originating a variety of different solutions. However the model based on *layers* is by far the most widespread. For this reason, through the paper we generally refer to this model, pinpointing the existence of alternative solutions where needed. Layers (Costanza and Hirschfeld, 2005) are a language abstraction which groups *partial method definitions* implementing behavioral fragments conceptually related to the same aspect of the application context. In Fig. 1 we show a simplified implementation of an adaptable storage implemented using COP concepts. This is a running example that we adopt throughout the paper. In the example and in the rest of the paper, where not explicitly said, we adopt a Java-like language, properly augmented with the features required for the explanation. Details not essential for the discussion, such as some type declarations or modifiers, are omitted. By calling the `getItem` method, the storage can be queried for a resource, which is by default searched on disk. The `getItem` method is partially redefined inside the `logLayer` layer and in the `cacheLayer` layer. The `logLayer` layer implements logging facilities and the `cacheLayer` layer adds a caching mechanism, which improves the response time. When `getItem` is called on an instance of the Storage class, if no layer is active the original version is executed, otherwise a partial definition in the active layer is executed.

Several solutions have been proposed in COP for layer activation. They are discussed in detail in Section 3. In the example of Fig. 1, the `with` keyword activates the given layers for the scoped block. As a convention, the last activated layer comes first in the execution. If more than one layer is active, the partial definitions are dynamically combined to get the resulting execution. For example if the `logLayer` layer and the `cacheLayer` are both active, logging and caching behaviors are obtained at the same time. Layers combination is achieved through the `proceed` keyword, which is similar to `proceed` in aspect oriented programming or `super` in object-oriented languages. Through `proceed`, the partial definition in the next active layer is executed. If no further partial definition is present in the active layers sequence, the original implementation is called.

In most COP languages layers are first class entities in the sense that they can be assigned to variables, passed as function parameters and returned as values. This is the fundamental way in which different parts of the program can communicate the adaptation to be performed.

In the paper we use the following conventions taken from Lincke et al. (2011): *layered method definition* are method definitions for which a *partial method definition* is present. These methods are dispatched according to the context-oriented semantics. Standard object-oriented method definitions are referred to as *plain method definition* and are not affected by the presence of layers.

For convenience, we list the existing COP languages in Table 1. For each language, we describe a distinguishing feature that characterizes it, and we refer to the literature for further study.

3. Flavors of context-oriented programming

In this section we present COP in more detail, discussing an overview of the features of the available implementations and the variations to the basic model described so far. We adopt the following approach. First we review the *implementation techniques* adopted for COP languages, which have a significant impact on their usage. For example, library-based implementations are easier to integrate with existing projects while ad hoc source-to-source compilers can be harder to be accepted in an established development system and can break tool compatibility. Then we consider the *dynamic* aspects of COP, i.e. how behavioral variations are activated. In many cases the `with`-based dynamically scoped activation is not adequate; for example due to the design of the application, it may be necessary to perform the activation on single objects rather than on control flows. We analyze the *static* aspect of COP, which is