Design and Implementation Constructs for the Development of Flexible, Component-Oriented Software Architectures

Michael Goedicke\(^1\), Gustaf Neumann\(^2\), and Uwe Zdun\(^1\)

\(^1\) Specification of Software Systems, University of Essen, Germany
\{goedicke,uzdun\}@cs.uni-essen.de
\(^2\) Department of Information Systems, Vienna University of Economics, Austria
gustaf.neumann@wu-wien.ac.at

Abstract Component-orientation is an emerging paradigm that promises components that are usable as prefabricated black-boxes. But components have the problem that they should be changeable and flexibly adaptable to a huge number of different application contexts and to changing requirements. We will argue, that sole parameterization – as the key variation technique of components – is not suitable to cope with all required change scenarios. A proper integration with multiple other paradigms, such as object-orientation, the usage of a scripting language as a flexible component glue, and the exploitation of high-level interception techniques can make components be easier (ex)-changeable and adaptable. These techniques can be applied without interfering with the component’s internals.

1 Introduction

The task of a software engineering project is to map a model of the real world (existing or invented) onto a computational system. The complexity and diversity of concrete real world systems can be overwhelming. This is no complexity in the algorithmic sense, but an complexity of an overwhelming amount of details and of particularities in the universe of discourse. By developing a model we reduce this complexity by finding and extracting commonalities. The key instruments of modeling are abstraction and partitioning. Analyses of commonalities let us understand the common elements of a targeted system. The aim of any analysis of commonalities is to group related members of a family, regardless whether the members are components, objects, modules, functions, etc.

Orthogonal to the task of modeling commonalities (where details are removed) is the task of engineering variability. It makes absolutely no sense to create abstractions to understand a family as a whole, if we do not introduce proper means for variation in the family members. Finding commonalities in software eases understanding and reduces the need for changes, while finding proper variabilities enables us to use the software at all, because we have to re-adapt the found abstractions to the concreteness of the modeled real world.
situation. Commonality and variability are competing concerns and its hard to find a proper balance between them by approaches that (a) model the real world from the scratch and then (b) try to reuse the common aspects in such upfront design models. The forces in the steps (a) and (b) can normally not be well integrated. We rather propose in this work to model only the interfaces and keep them variable. Techniques for “programmable interfaces” let us flexibly glue the application parts together.

There are recurring ways for finding good abstractions and partitionings. “Good” means that they provide a tenable amount of commonalities to let us understand the problem and produce long-lasting software, but still enable us to easily introduce (expected and unexpected) changes. Such patterns of organizing abstraction around commonalities and variations are popularly called “paradigms”. In software engineering a paradigm is a set of rules for abstraction, partitioning, and modeling of a system. E.g., the object-oriented paradigm structures the design/program around the data, but focuses on behavior [23]. It allows us to introduce variations in data structures/connections and algorithm details. Each paradigm has a key commonality and variation.

If we implement a system, we have to deal with a broad variety of paradigms. Coplien [3] discusses the need for multi-paradigms. In fact nearly any good real world software system is designed and implemented using multiple paradigms, simply because nearly no complex real situation exists, that can be described with one paradigm sufficiently. E.g., in nearly every large C++ program a mixture of object-oriented, procedural, template, and various outboard paradigms exists. Here, outboard paradigm [3] means a paradigm that is not supported by the programming language itself, but by a used technology, like the relational paradigm adopted from a relational database.

In the focus of this paper are language constructs and concepts for design and implementation that overcome current problems of the component- and object-oriented paradigms and their integration. Firstly, we will discuss these paradigms and their current integration problems. Afterwards we present some language concepts of the language XOTcl: Firstly we will discuss concepts which can be mapped manually to current mainstream languages, then we will present some interception techniques that are missing in current mainstream languages. Finally we will generalize our approach and compare to related work.

2 Combination of Component- and Object-Oriented Paradigm

2.1 Component-Oriented Paradigm

The very idea of component-based development is to increase productivity of building software systems, by assembling prefabricated, widely-used components. Components are self-contained, parameterizable building blocks with explicit interfaces. Component-based development aims at the replaceability of components and the transferability of components to a different context, thus enabling component reuse.