Chapter 11
Beyond SOA and REST for Distributed Application Integration

José Carlos Martins Delgado
Universidade de Lisboa, Portugal

ABSTRACT

The fundamental problem of distributed application integration is reducing application coupling as much as possible while still meeting the minimum interoperability requirements. Service-oriented architecture (SOA) and representational state transfer (REST) are the most used architectural styles to deal with this problem. This chapter performs a comparative study of these styles and shows that, while both solve basic interoperability, neither of them minimizes coupling, since data description schemas are shared by the interacting applications (symmetric interoperability). SOA is oriented towards behavior (services) and REST towards state (structured resources). Services have no structure and resources have a fixed service. This chapter proposes a new architectural style, structured services, that combines the best characteristics of SOA and REST (services can have structure and resources can implement application-specific services), while using asymmetric interoperability (schema compatibility is based on structural compliance and conformance) to minimize application coupling.

INTRODUCTION

A software application is usually complex and programming it involves modeling the entities of the problem domain by modules that need to interact to achieve the goals that justify the purpose of that application. In addition, the context and specifications of the problem are dynamic and typically change frequently. Complexity and


Copyright © 2018, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
changeability have always been a huge challenge for software developers, affecting not only applications but also virtually all aspects of computer science.

The object-oriented modeling paradigm, in particular through the Unified Modeling Language (UML) (Dennis, Wixom, & Tegarden, 2015), has long been recognized as an adequate solution to these difficulties. The main tenet is to divide and conquer complexity by structuring programs into objects, according to principles that tend to limit the impact of a change, by reducing dependencies between objects, and allowing them to be incrementally adapted through inheritance. Table 1 describes the main relevant principles.

Ideally, one would like to extend these principles to interacting objects in a distributed environment. Unfortunately, most of the advantages of the object-oriented paradigm, namely inheritance and polymorphism, stem from the fact that the lifecycles of the objects are coupled, either at compile or link time, which is not the case for distributed applications. These are loosely coupled and evolve independently, but need nonetheless to interact and cooperate towards some common goals. As one application changes, all the others that use it must also be adapted, in order to maintain interoperability.

There is no universally accepted definition of interoperability, since its meaning can vary accordingly to the perspective, context or domain under consideration. Nevertheless, the ISO/IEC/IEEE 24765 standard (ISO, 2010) provides the seemingly most cited definition of interoperability: “the ability of two or more systems or components to exchange information and to use the information that has been exchanged”.

Table 1. Main principles of object-oriented modeling

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Least semantic gap</td>
<td>The entities in the problem and solution spaces should have a one to one mapping, by modeling each problem entity with a solution entity, with both state and related behavior, and corresponding structures.</td>
</tr>
<tr>
<td>P2</td>
<td>Information hiding</td>
<td>There must be a clear separation between interface (syntax and semantics) and its implementation. This reduces the change propagation effect. A change in one object affects those that use it only if its interface is affected.</td>
</tr>
<tr>
<td>P3</td>
<td>Substitution</td>
<td>An object X should be able to be used with its own implementation wherever an object Y is expected, as long as X conforms to Y, a form of polymorphism.</td>
</tr>
<tr>
<td>P4</td>
<td>Non-duplication of information</td>
<td>Objects with similar features (state or behavior) should factor these features and share a common definition, usually implemented by inheritance, instead of duplicating it.</td>
</tr>
<tr>
<td>P5</td>
<td>Open-closed</td>
<td>An object should be closed, so that it can be used, but at the same time open, so that it can be changed without having to close it. This means that it is possible to tune up objects by redefining operations, even if the source code is not available.</td>
</tr>
</tbody>
</table>
Knowledge Management and Entrepreneurship Cases in India
[www.igi-global.com/chapter/knowledge-management-entrepreneurship-cases-india/46857?camid=4v1a](www.igi-global.com/chapter/knowledge-management-entrepreneurship-cases-india/46857?camid=4v1a)

Anchoring Corporate Innovation Communities in Organizations: A Taxonomy
[www.igi-global.com/article/anchoring-corporate-innovation-communities-organizations/61425?camid=4v1a](www.igi-global.com/article/anchoring-corporate-innovation-communities-organizations/61425?camid=4v1a)