Chapter 2

Structural Interoperability as a Basis for Service Adaptability

José C. Delgado

Instituto Superior Técnico, Technical University of Lisbon, Portugal

ABSTRACT

Web Services appeared essentially as an interoperability solution and REST as a closer match to the semantics of protocols such as HTTP. Clearly influenced by the original browsing goals of the Web, these technologies are not native solutions to the service-oriented paradigm, exhibit limitations to interoperability, and behavior has to be implemented in a separate language. Web Services offer a WSDL document to describe them, but assume that complex data use the same schema in both interacting services, which increases their coupling. This chapter discusses interoperability, from the perspective of both the consumer (compliance) and provider (conformance) services, and it argues that compliance is a weaker requirement for service interoperability than conformance and should be the cornerstone to decrease coupling and to favor adaptability. Structural interoperability is used, given that the life-cycles of distributed resources are decoupled. Metrics to quantify adaptability, based on similarity and decoupling, are proposed.

INTRODUCTION

The complexity and changeability of software have long been a huge challenge for software developers, affecting not only applications but also virtually all aspects of computer science, such as formats, protocols, languages, platforms, techniques and standards. It seems that change and adaptation are really the only constant things.

The object-oriented paradigm appeared as a solution to these issues, by structuring the programs according to some principles that tend
to limit the impact of a change, affecting fewer objects, and to increase the adaptability, allowing resources to be adapted, even without the source code (Meyer, 2000):

- **The Least Semantic Gap Principle**: 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;

- **The Information Hiding Principle**: There must be a clear separation between interface (syntax and semantics) and its implementation. This reduces the change propagation effect. A change in one resource affects those that use it only if its interface is affected;

- **The Substitution Principle**: 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;

- **The Non-Duplication of Information Principle**: Objects with similar features (state or behavior) should factor them and share a common definition, usually implemented by inheritance, instead of duplicating it;

- **The Open-Closed Principle**: An object should be closed, so that it can be used, but at the same time open, so that it can be changed. Actually, this is a corollary of the two previous principles and it means that it is possible to tune up objects by redefining operations, even if the source code is not available.

Unfortunately, most of the advantages of the object-oriented paradigm, namely information sharing (inheritance, in practice) 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 services, in SOA and cloud environments. These are loosely coupled and evolve independently, but need nonetheless to interoperate. As one service changes, those that use it must adapt to maintain interoperability.

Figure 1 illustrates the possibilities of service adaptability. A client application can invoke several services. Its context consists of the platform that executes it, which can be a device such as a server, a laptop, a tablet or a smart phone, and the services available to it. Each service needs a platform to execute it and its environment is the set of services that it can invoke. For simplicity, the platform and environment is shown for service B only. The client application can also be a service, available to other applications.

The need for adaptation, either at design/compile time or at runtime, can arise in situations such as:

- Context awareness (Hong, Suh & Kim, 2009), both for applications and services, which involves two main slants:
  - Platform awareness, according to the platform’s characteristics and capabilities, either from the local application or service or the services they invoke (Ortiz & Prado, 2009).
  - Environment awareness, in which the set of services available can vary ac-