Chapter 17
The Role of Compliance and Conformance in Software Engineering

José C. Delgado
Instituto Superior Técnico, Universidade de Lisboa, Portugal

ABSTRACT

One of the most fundamental aspects of software engineering is the ability of software artifacts, namely programs, to interact and to produce applications that are more complex. This is known as interoperability, but, in most cases, it is dealt with at the syntactic level only. This chapter analyzes the interoperability problem from the point of view of abstract software artifacts and proposes a multidimensional framework that not only structures the description of these artifacts but also provides insight into the details of the interaction between them. The framework has four dimensions (lifecycle, concreteness level, concerns, and version). To support and characterize the interaction between artifacts, this chapter uses the concepts of compliance and conformance, which can establish partial interoperability between the artifacts. This reduces coupling while still allowing the required interoperability, which increases adaptability and changeability according to metrics that are proposed and contributes to a sustainable interoperability.

INTRODUCTION

Software systems are neither monolithic nor self-contained, but rather composed of models, specifications and working modules that are interrelated and need to fit together, usually by design. Decomposing a complex problem into several simpler and smaller artifacts, in a divide & conquer approach, is a fundamental software engineering technique to deal with complexity and improve design characteristics such as reusability, agility, changeability, adaptability and reliability.

An artifact can be any entity related to software engineering such as a concept, a specification or a program. The relationships between artifacts are essential to accomplish the goals of the software system but, at the same time, they create dependencies and coupling between them that translate into constraints and partially hinder these characteristics.

DOI: 10.4018/978-1-4666-6026-7.ch017

Therefore, software engineering can be described as the application of engineering principles, methods and techniques to computer-based artifacts under *quality* and *sustainability* constraints. Quality means that the problem needs to be decomposed into the right artifacts and with the right relationships (satisfying the problem's specifications with a good architecture). Sustainability (Jardim-Goncalves, Popplewell & Grilo, 2012) means that changes in the problem specification or in its context should translate to incremental changes in the artifacts and their relationships, implemented at a faster rate than the changes that motivated them.

Quality and sustainability are not exclusive of software engineering. A car, for example, is a system with several thousand components that need to fit together perfectly, under the same constraints. What distinguishes software engineering is the fact that, in most cases, artifacts are virtual (easy to create and to destroy), very flexible and exhibit a high variability rate. A computer program can be changed in minutes or even seconds, which is certainly not the case of physical products such as cars.

This chapter concentrates on the sustainability side of software engineering and specifically in the relationships between software artifacts, in an attempt to improve the characteristics mentioned above. The basic tenets that we will use are:

- If an artifact *A* has no relationship with an artifact *B* (does not depend on it), then *B* can change freely without impacting *A*. This is good for sustainability. Ideally, all artifacts should be completely independent (decoupled from all other artifacts);
- Artifacts that have no relationship cannot cooperate, which means that no value comes from decomposing a system into artifacts. Any system needs that artifacts establish relationships and cooperate, somehow. This implies some coupling between some artifacts.

These are conflicting goals. The fundamental problem that we are trying to solve is how to get the best compromise, or *how to minimize coupling as much as possible while still satisfying the problem's specifications*.

Relationships between software artifacts can be established at various levels, such as:

- *Conceptual*, involving concepts such as strategies, goals and architectures. For example, different artifacts may cooperate towards some common goal or complementary goals;
- *Documental*, which pertains mainly to specifications. For example, a given artifact must use the features defined by some standard;
- *Design*, entailing the way artifacts are used to build a composed system. For example, any software development method will include a decomposition of the problem's specification and a composition of already existing artifacts (such as a software library), trying to match both approaches;
- *Operational*, in which working artifacts (such as software modules) interact by sending messages. The receiver of a message must be able to understand the content of a message and the intention of the sender in sending that message.

This means that relationships between artifacts are not limited to message based interaction but can occur at any stage of the artifacts' lifecycle, right from their conception, even if the artifact never becomes active and able to interact, such as a concept or a document.

At the operational level, in particular, it is also important to check whether artifacts can be bound together in a single application or are distributed, most likely in different computers and probably implemented in different programming languages. Solutions to support the interaction between these artifacts are different in both cases. The concept