Chapter XIV

Complexity-Based Evaluation of the Evolution of XML and UML Systems

Ana Isabel Cardoso, University of Madeira, DME, Portugal
Peter Kokol, University of Maribor, FERI, Slovenia
Mitja Lenic, University of Maribor, FERI, Slovenia
Rui Gustavo Crespo, Technical University of Lisbon, DEEC, Portugal

Abstract

This chapter analyses current problems in the management of software evolution and argues the need to use the Chaos Theory to model software systems. Several correlation metrics are described, and the authors conclude the Long-Range Correlation looks to be the most promising metrics. The Long-Range Correlation measures for XML and Java files are very similar. We then identify the number of ideas that may be raised in the process of software development, and link the different behaviours of the software evolution to the Verhulst model. Finally, we analyse one industrial test case and verify that the behaviours of software evolution are represented in the Verhulst model.
Introduction

Since the 1960s, software engineers have recognised the growing costs, errors, and delays in the development and evolution of software systems are manifestations of a **software crisis** (Naur & Randell, 1969). The impact of the software crisis is huge. For example, the annual costs of software defects in the US are estimated to be up to $60 billion (NIST, 2002).

Software systems are frequently updated and, nowadays, the relative cost for maintaining the software evolution is around 90% (Seacord, 2003). Software maintenance is now an important issue, both in industry and research institutions (Grubb & Takang, 2003). The pressures for changing software systems have been identified and include (a) the technology, market, and legislation evolution, (b) the need to correct undetected errors, and (c) the increase of understanding of the system by the users after the release installation (Lehman & Belady, 1985). As result of the pressures for system change, either the software system adapts to these changes or becomes less useful until, ultimately, it must be discarded. After analysing multiple versions of the IBM OS/360 operating system, Lehman has suggested that the evolution of a software system is subject to three informal laws: (a) **continuing change**, that states a program used in a real-world environment must change, (b) **increasing entropy**, that prescribes the program structure becomes more complex unless efforts are made to avoid the complexity, and (c) **statistically smooth growth**, that enounces the global system metrics appear locally stochastic in time and space but are self-regulating and statistically smooth (Lehman, 1980).

Nowadays, software systems have a wide range of applications, such as reactive systems for Machine Control, symbol processing in Artificial Intelligence, and number crunching for Simulation. Developers now have available many design and programming languages, each one oriented for a specific range of applications. The object-oriented design and programming languages have been adopted for the development of many applications because they are closer to the real world, and data are not shared, which reduces overall system coupling as there is no possibility of unexpected modifications to shared information. The Unified Modeling Language (UML) (Fowler & Kendall, 1999) and Java (Arnold & Gosling, 1997) are two examples of widely used object-oriented design and programming languages.

Project managers recognise that the higher the semantic organisation of the computer applications is, the easier the software understanding and the lower the cost of system evolution. Informally, the semantic organisation of a program is given by the regularity presence of software elements. Object-oriented systems enforce regularity in some of the software elements, such as data encapsulation and class hierarchies. Our research focuses on the regularity of the software elements in UML diagrams and Java programs that vary according to the programmers and to the project goals. For example, programmers with different practices create, in the UML sequence diagrams, different sequences of message interactions between the same classes for the same software system.

Many project teams and researchers have proposed metrics to quantify and control the software development and evolution (Fenton & Pfleeger, 1997). The results, however, fell below the initial expectations. One possible reason for this failure is that the current