Chapter IX

The STECC Framework: An Architecture for Self-Testable Components

Sami Beydeda, Federal Finance Office, Germany

Abstract

Development of a software system from existing components can surely have various benefits, but can also entail a series of problems. One type of problem is caused by a limited exchange of information between the developer and user of a component. A limited exchange and thereby a lack of information can have various consequences, among them the requirement to test a component prior to its integration into a software system. A lack of information cannot only make testing prior to integration necessary; it can also complicate this task. However, difficulties in testing can be avoided if certain provisions to increase testability are taken beforehand. This article briefly describes a new form of improving testability of, particularly commercial, components, the self-testing COTS components (STECC) strategy and explains in detail the STECC framework, which implements the necessary technical architecture to augment Java components with self-testability.
Introduction

A major trend in software engineering is that of component-based development. The underlying idea of component-based development is to use existing components for the construction of a software system and to construct it by integrating them instead of programming the software system from scratch. This idea is neither new in the construction of complex systems in general nor in software engineering in particular. In other, more mature engineering disciplines, such as the construction of electronic devices, it is a usual procedure to use existing components. In software engineering, the use of components in software development was suggested more than 30 years ago (McIlroy, 1968).

The expected benefits of using components in software development and the motivations for component-based development are manifold. Among others, components are expected to improve the success factors of software-development projects, such as quality of the system developed and adherence to time and budget restrictions, similar to development projects in other disciplines. However, software components strongly differ from components such as those used in the construction of electronic devices. Software components generally have much more inherent complexity. Appropriate methods to cope with this complexity still do not exist, which gives an indication of the maturity of this paradigm and of software engineering as a whole. Even if some success is achieved with the use of components, component-based development still has to show its full potential.

As one of the success factors, the use of components was assumed to have positively affected the quality of the software system developed. Components were assumed to have reached a high level of quality in a short period of time, for instance, due to market pressure. Also, software systems consisting of such components were expected to inherit this high level of quality. Unfortunately, experience showed that this assumption does not necessarily hold in practice. One reason is a lack of information (Beydeda & Gruhn, 2003b; Beydeda & Gruhn, 2003c), particularly in the case of commercial components. The provider of a component might not be able to anticipate all possible application contexts in which the component might be used. The same also applies to technical environments in which the component might be embedded. Assumptions in the development of a component concerning, for instance, the application environment might be inaccurate or even wrong and quality assurance conducted by the component provider might not be effective. The component might exhibit failures.

A lack of information can also concern the user of a component. The user might also suffer from such a lack insofar that the component might not be sufficiently documented due to inaccurate or incorrect information from the provider with regards to the required documentation. This obviously can have various adverse effects on development and maintenance of a system using the component. Context-dependent tests by the component provider and insufficient documentation often obligate the component user to test a component prior to its use in the development of a software system. A problem, again due to a lack of information, might be encountered when failures are revealed in such tests. The component user usually has to rely on the component provider to remove the faults causing the failures, which is one of the forms in which the component user’s
Related Content

Formalization of UML Composition in OCL
[www.igi-global.com/article/formalization-uml-composition-ocl/77616?camid=4v1a](www.igi-global.com/article/formalization-uml-composition-ocl/77616?camid=4v1a)

Watermarking Scheme with CS Encryption for Security and Piracy of Digital Audio Signals

Integrating Compliance Management in Service-Driven Computing: Conceptual Models and Automation Architecture
[www.igi-global.com/chapter/integrating-compliance-management-in-service-driven-computing/115439?camid=4v1a](www.igi-global.com/chapter/integrating-compliance-management-in-service-driven-computing/115439?camid=4v1a)

Evaluation of the Challenges of Developing Secure Software Using the Agile Approach
[www.igi-global.com/article/evaluation-of-the-challenges-of-developing-secure-software-using-the-agile-approach/144788?camid=4v1a](www.igi-global.com/article/evaluation-of-the-challenges-of-developing-secure-software-using-the-agile-approach/144788?camid=4v1a)