Article Summaries

Pages: p. 4

Software Architecture

The Golden Age of Software Architecture

by Mary Shaw and Paul Clements, pp. 31–39. This retrospective on nearly two decades of research examines the software architecture field's maturation by tracing the evolution of its research questions and results. Early qualitative results set the stage for later precision, formality, and automation. Results over the ensuing decades have matured and moved into practice.

In Practice: UML Software Architecture and Design Description

by Christian F.J. Lange, Michel R.V. Chaudron, and Johan Muskens, pp. 40–46. To determine how UML is being used in current software architecting and design, the authors surveyed practitioners and analyzed case studies of industry projects. Their results show that UML is used rather loosely and that UML models are often incomplete. This leads to miscommunication and other implementation and maintenance problems. The authors conclude with recommendations and techniques for controlling UML model quality.

Software Architecture-Centric Methods and Agile Development

by Robert L. Nord and James E. Tomayko, pp. 47–53. Including architecture-centric design and analysis methods in the Extreme Programming framework can help software developers address quality attributes in an explicit, methodical, engineering-principled way. Properly managed, architecture-centric methods can be a cost-effective addition to the software development process and will increase system and product quality.

Using Architectural Patterns and Blueprints for Service-Oriented Architecture

by Michael Stal, pp. 54–61. Using software patterns and blueprints to express a service-oriented architecture's fundamental principles supports the efficient use of SOA technologies for application development. Software patterns can express almost all architecture principles that span the space of SOA technologies. This architecture-centric approach helps developers understand service-oriented infrastructures and build SOA applications that meet operational and developmental requirements.

Using Architecture Models for Runtime Adaptability

by Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven, pp. 62–70. Recently, the introduction of software platforms supporting component plug-in and dynamic binding has facilitated adaptation of software systems at runtime. Preserving the properties described by the architecture model during adaptation is an important task. The authors propose a self-adaptation approach that exploits architecture models to reason about and control adaptation at runtime. They can derive runtime models from design models facilitating the developers' task. They developed the approach in the context of mobile computing.

Architecture Description Languages for High-Integrity Real-Time Systems

by Alek Radjenovic and Richard Paige, pp. 71–79. Safety-critical systems, also known as high-integrity real-time systems, require architecture description languages that model the entire system rather than specific parts or aspects of it. ADLs pose unique challenges in their construction. Architecture Information Modeling is an ADL developed in collaboration with the safety industry. AIM provides language

flexibility, abstraction layering, custom view creation, a design-by-view process, and safety and change control. A case study demonstrates the application of the AIM concepts in a top-down software design.

A Fault-Tolerant Architectural Approach for Dependable Systems

by Rogério de Lemos, Paulo Asterio de Castro Guerra, and Cecília Mary Fischer Rubira, pp. 80–87. Developers typically address dependability concerns in the late phases of system development. However, two trends are compelling developers to consider dependability earlier, at the architectural level. First, emerging applications are increasingly complex. Second, to address this first trend, developers are increasingly attempting to build dependable systems from existing undependable components. A new architectural approach employs exception handling to represent and analyze fault-tolerant software systems. It partitions architectural elements into normal and exceptional parts, thus promoting a clear separation of concerns regarding how to detect and handle errors.

Features

Emphasizing Human Capabilities in Software Development

by Silvia T. Acuña, Natalia Juristo, and Ana M. Moreno, pp. 94–101. The human dimension is a critical factor in organizations, especially in software companies where the production process is essentially intellectual. One key aspect of the software engineering workforce is identifying the people best suited for development roles. Despite its importance, there is little support for this task, leading to unease and misunderstandings among the people involved. A capability-based procedure can aid managers at small to medium-sized software organizations.

Coupling Metrics for Ontology-Based Systems

by Anthony M. Orme, Haining Yao, and Letha H. Etzkorn, pp. 102–108. Measuring system coupling is a commonly accepted software engineering practice associated with producing high-quality software products. In many application domains, we can assess coupling in ontology-based systems before system development by measuring coupling in ontology data. A proposed set of metrics measures coupling of ontology data in ontology-based systems represented in the Web Ontology Language, a derivative of XML. A real-world study demonstrates the metrics' use in integrating ontologies.

Software architecture and software design are two aspects of the same topic. Both are about how software is structured in order to perform its tasks. The term "software architecture" typically refers to the bigger structures of a software system, whereas "software design" typically refers to the smaller structures. Exactly where the boundary is between architecture and design is hard to say, since the architecture of a system also affects its design. Software Architecture Design is a crucial step for software and application developers to describe the basic software structure by dividing functional areas into layers. It depicts how a typical software system might interact with its users, external systems, data sources, and services. Software architecture is usually designed into four layers (some also make it three), which are, from top to bottom, presentation layer, service layer, business layer, and data layer. Software Architecture from University of Alberta. The way that software components — subroutines, classes, functions, etc. — are arranged, and the interactions between them, is called architecture. In this course you will study the ways these

About this Course. Software Architecture and Design teaches the principles and concepts involved in the analysis and design of large software systems. This course is split into four sections: (1) Introduction, (2) UML and Analysis, (3) Software Architecture, and (4) Software Design. Course Cost. Free. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment. We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In Architecture, nonfunctional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished. Software Architecture. Architecture serves as a blueprint for a system.