article summaries

MARCH/APRIL 2005, VOLUME TWENTY-TWO, NUMBER TWO

POSTMODERN SOFTWARE DESIGN

Architecture Decisions: Demystifying Architecture
by Jeff Tyree and Art Akerman, pp. 19–27. Bridging the gap between requirements and the right technical solution involves a lot of magic. The authors explain how explicitly documenting major architecture decisions makes the architecture development process more structured and transparent. Additionally, it clarifies the architects’ rationale for stakeholders, designers, and other architects. Architecture decisions are also a helpful addition to more traditional architecture approaches.

An Ontology for Microarchitectural Design Knowledge
by Javier Garzás and Mario Piattini, pp. 28–33. A thorough grasp of object-oriented microarchitectural design knowledge is crucial to software engineering. The authors have developed an ontology that facilitates this task. The ontology considers not only hot topics such as refactorings and patterns but also important elements of knowledge such as heuristics, principles, best practices, and “bad smells.” Thus, it can help develop unified catalogs of knowledge, structure and unify accumulated knowledge, improve communication, teach concepts and their relationships, share knowledge, and resolve incompatibilities in terminology.

Architecture Reviews: Practice and Experience
by Joseph F. Maranzano, Sandra A. Rozsy- pal, Guy W. Warnken, David M. Weiss, Patricia E. Wirth, and Gus H. Zimmerman, pp. 34–43. Architecture reviews have evolved over the past decade to become a critical part of continuing efforts to improve the software development process and the systems it produces. The reviews identify project problems before they become costly to fix and provide timely information to upper management so that they can make better-informed decisions. Development managers who want to establish architecture reviews as part of their development methodology will find that the process isn’t difficult to implement, especially with experienced domain experts and a willing management team.

Model Mapping Using Formalism Extensions
by Guy Caplat and Jean-Louis Sourrouille, pp. 44–51. The OMG’s Model Driven Architecture separates system specification from implementations in platform-independent models and platform-specific models, respectively. Software development involves a sequence of model mappings from an initial PIM to a final PSM precise enough for direct translation into executable code. The mapping process involves many intralanguage mappings and at least one interlanguage mapping from a semiformal to a formal language. Using formalism extension mechanisms to manage the “semantic gap” in the interlanguage mapping process establishes a clear methodology for PIM-to-code transformation.

Casting Software Design in the Function-Behavior-Structure Framework
by Philippe Kruchten, pp. 52–58. Using John Gero’s Function-Behavior-Structure framework as a reference, the author contrasts the software engineering process with processes in other engineering disciplines. He notes that software engineers use certain terms, such as design, analysis, and documentation, in significantly different ways from other engineers—in particular, that “design” in software engineering is more limited in scope than in other fields. Having a broader understanding of other engineering disciplines—and software engineering’s place among them—can benefit software engineers involved in multidisciplinary projects.

FEATURES

Hypermedia Systems Development Practices: A Survey
by Michael Lang and Brian Fitzgerald, pp. 68–75. Do shoddy project management, inadequate requirements analysis and planning, and ad hoc “quick and dirty” techniques really characterize hypermedia development? How are such projects guided and controlled? Although the literature alleges a hypermedia development “crisis,” this survey shows that talk of a crisis is largely unfounded.

Automatic Test Case Optimization: A Bacteriologic Algorithm
by Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon, pp. 72–82. The quality of test cases is an important factor in estimating confidence in a component under test. One way to evaluate this quality is by mutation analysis, which measures the proportion of seeded faults that the test cases detect. While generating a basic set of test cases can be easy, improving its quality might require prohibitive effort. A bacteriologic algorithm, adapted from genetic algorithms and inspired by evolutionary ecology, can automate test optimization. Test generation for a C# parser illustrates the approach.