Capturing Design Knowledge

Uwe Zdun, *Vienna University of Technology*

Capturing software design knowledge is important because it tends to evaporate as software systems evolve. This has severe consequences for many software projects. Knowledge evaporation is a problem for all kinds of design knowledge, including software architectural knowledge, object-oriented design knowledge, knowledge about implementation details, and knowledge about the software process. We can ascribe many of the reasons for this to two "laws" of software evolution: increasing complexity and continuing change. The sidebar "Design Knowledge Evaporation and the Laws of Software Evolution" summarizes these laws and explains their relationship to design knowledge evaporation.

To counteract this phenomenon, effective, systematic documentation of design knowledge is important. However, many proposed approaches for capturing design knowledge are still experimental or in an early-adoption stage. In practice, many developers, designers, and architects consider the capture of design knowledge a resource-intensive process without tangible, short-term gains. They often address it inadequately or even skip it entirely. On the other hand, some do use a set of accepted approaches to deal with parts of the knowledge evaporation problem. For more information, see the sidebar "Current and New Approaches for Capturing Design Knowledge."

For this minitheme, the three articles we selected from our regular queue address increasing complexity and continuing change by providing explicit means for capturing design knowledge.

"Modularization of a Large-Scale Business Application: A Case Study," by Santonu Sarkar, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan, and Saravanan Sivagnanam, reports on using a subset of the common approaches for capturing design knowledge. The project deals with reengineering for better modularity. The article's starting point is the situation that large software systems deteriorate into unmanageably complex monoliths because continuing changes aren't applied systematically. Among the approaches used for capturing design knowledge are guidelines on how to structure modules

0740-7459/09/$25.00 © 2009 IEEE

Design Knowledge Evaporation and the Laws of Software Evolution

When Meir Manny Lehman and Laszlo A. Belady studied the historical data of successive releases of large operating systems, they found properties of software evolution processes that they postulated as "laws of software evolution."[1] The two most prominent laws are *increasing complexity* (entropy) and *continuing change*.

The law of increasing complexity states that a software system becomes more complex as it evolves and that extra resources are needed to preserve and simplify its structure. So, a major goal of software design is to avoid increasing complexity. If a large design isn't planned, it's often difficult to maintain an overview of the system as a whole. The system grows through quick bug fixes and feature enhancements, gradually losing design knowledge about the system. Consequently, the complexity gets out of hand. Retroactively making a design manageable usually requires major reengineering at enormous cost.

The law of continuing change states that a software system used in a real-world environment must change or become progressively less useful in that environment. So, a major goal of software design is to increase flexibility. If a design isn't flexible enough, it might not be able to cope with change requests. If a design doesn't properly support flexibility, quick fixes are often applied to solve problems in the course of daily business. With each quick fix, however, the design gets more complex and less flexible, and design knowledge evaporates. That is, it becomes progressively harder to fix the design problems that have caused the missing flexibility.

Thus, the design problems of complexity and flexibility affect each other and, if not solved, lead to design knowledge evaporation. If complex systems have no systematic way to record and share design knowledge, that knowledge gradually evaporates as the system evolves. The result is an even greater increase in complexity and longer change cycles. Eventually, the system gets so complex and hard to change that even simple fixes and feature enhancements are excessively time consuming and costly.

Reference

1. M.M. Lehman and L.A. Belady, *Program Evolution—Processes of Software Change*, Academic Press, 1985.

About the Author

Uwe Zdun is an assistant professor in the Distributed Systems Group at the Vienna University of Technology. His research interests include software patterns, software architecture, language engineering, service-oriented architecture, distributed systems, and object orientation. He authored or coauthored Frag, Extended Object Tcl (XOTcl), Leela, ActiWeb, and many other open source software systems. He's coauthor of *Remoting Patterns: Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware* (John Wiley & Sons) and *Software-Architektur: Grundlagen, Konzepte, Praxis* (Software Architecture: Basics, Concepts, Practice; Elsevier/Spektrum). He's also the European editor of the *Transactions on Pattern Languages of Programming*. Contact him at zdun@infosys.tuwien.ac.at.

and module interactions, explicit modeling of the system's modularization, tools to gather the design knowledge and aid modularization, involvement of stakeholders other than developers, and metrics to measure the quality of the modularization.

In "The Decision View's Role in Software Architecture Practice," Philippe Kruchten, Rafael Capilla, and Juan Carlos Dueñas propose augmenting architectural view models using a decision view that extends the traditional views with information on the design rationale. They argue that a decision view can help record and document the key decisions with an acceptable overhead. The approach is based on the well-known 4+1 view model but can easily be extended to other such models. The article proposes a lightweight approach that architects can apply as an extension to existing view models. In addition, the authors describe the historical evolution of software architecture representation.

Finally, "Software Architecture Design Reasoning: A Case for Improved Methodology Support," by Antony Tang, Jun Han, and Rajesh Vasa, proposes a novel approach to modeling design decisions and supporting reasoning about software architecture designs. Explicitly modeling design decisions makes architectural designs justifiable through the design rationale, and support for design reasoning enables the analysis of causal relationships among the design context, the design justification, and the design result. The authors suggest that this eases understanding, analysis, and communication of the architecture and hence reduces the complexity with which architects must deal. In addition, it supports changing the design: we can analyze the impact of changes when a system evolves while still accessing the original design rationale. This article targets readers who are interested in more experimental ways to capture and use design knowledge.

Depending on the situation, all three approaches described in these articles might help address the problems that many projects face: increasing complexity and continuing change.

Current and New Approaches for Capturing Design Knowledge

In practice, documenting knowledge is often considered a resource-intensive process without tangible, short-term gains, so quite often it's skipped or performed inadequately.[1,2] Many current approaches for capturing design knowledge are still relatively novel, so many industry projects prefer to use just a few particular existing methods and techniques. However, several practical ways to share design knowledge are already widely used. Here's a nonexhaustive list:

- establishing a clear process and guidelines for capturing design knowledge,
- modeling the central design-knowledge artifacts explicitly, using formal or semiformal models (such as UML diagrams),
- adding the design rationale and design-decision documentation to the system documentation,
- modeling close to the software system's domain (for example, banking, in the case of the article by Santonu Sarkar and his colleagues in this issue) to share knowledge more easily with nontechnical stakeholders,
- using model-driven software development or similar techniques to generate code from the formal or semiformal models,
- including stakeholders other than developers and designers in capturing design knowledge to facilitate knowledge exchange,
- using existing tools to analyze and reverse-engineer the design knowledge in a system,
- using quantitative and qualitative measures to analyze and assess a system to be able to make informed design decisions,
- establishing management (and other relevant stakeholder) support for knowledge capture to justify the costs involved, and
- using software patterns or other approaches that help minimize the effort involved in knowledge capture.[2]

The approach taken by Sarkar and his colleagues in this issue uses a subset of these approaches.

The field of capturing design knowledge includes many topics that existing approaches haven't yet fully covered. Examples include

- modeling of design knowledge,
- notations for capturing and visualizing design knowledge,
- teaching about design knowledge and methods to capture it,
- communicating design knowledge in a team,
- tools to support design knowledge representation and exchange,
- software patterns as a way to represent design knowledge, and
- traceability between different kinds of design knowledge.[3,4]

Many of these topics still contain numerous open issues or are in an experimental or early adoption phase.

The challenge of getting such approaches accepted in mainstream practice is that professorial designers often make design decisions unconsciously.[2] Often, the designers documenting the design knowledge don't consider retroactive knowledge documentation to offer short-term benefits for their work.[1] So, additional benefits for the designers, such as the design-reasoning support in the approach described by Antony Tang, Jun Han, and Rajesh Vasa in this issue, could help justify the extra effort. Lightweight approaches such as the decision view proposed by Philippe Kruchten, Rafael Capilla, and Juan Carlos Dueñas in this issue can help lower the entry barrier to applying techniques for capturing explicit design knowledge.

References

1. O. Zimmermann et al., "Combining Pattern Languages and Architectural Decision Models in a Comprehensive and Comprehensible Design Method," *Proc. 7th Working IEEE/IFIP Conf. Software Architecture* (WICSA 08), IEEE CS Press, 2008, pp. 156–166.
2. N. Harrison, P. Avgeriou, and U. Zdun, "Using Patterns to Capture Architectural Decisions," *IEEE Software*, vol. 24, no. 4, 2007, pp. 38–45.
3. P. Lago and P. Avgeriou, "First Workshop on Sharing and Reusing Architectural Knowledge," *SIGSOFT Software Eng. Notes*, vol. 31, no. 5, 2006, pp. 32–36.
4. P. Avgeriou, P. Lago, and P. Kruchten, "Sharing and Reusing Architectural Knowledge (Shark 2008)," companion to *Proc. 30th Int'l Conf. Software Eng.* (ICSE 08), IEEE CS Press, 2008.

IEEE Computer Society Members SAVE 25%

on all conferences sponsored by the IEEE Computer Society

www.computer.org/join