

## A challenge of modeling how to use an architecture

Tarja Systä

Tampere University of Technology

In an ideal case, software systems are built by combining existing software components with well defined and clear interfaces in a “plug and play” fashion. However, this seldom is the case in reality. Instead, various kinds of components available typically do not even plug together, or if they do they do not play. There are several reasons for the lack of interoperability. For instance, one of the main obstacles in integrating COTS tools is that they typically support only one-way information exchange; there is no or minimal support for mechanisms that allow the system to notify the component or support for data synchronicity [EB01]. The new component and internet-based technologies have increased not only the challenges for architecture recovery but also for software architecting. Design patterns [GHJV94], COTS components, application frameworks etc. are examples of large size software building blocks. In practice, constructing a systems architecture from components of larger granularity unfortunately requires a lot of glue code (that is not limited to data sharing or procedural calls). Furthermore, net-centric software systems are typically more vulnerable to run-time problems than traditional systems that run on a single machine. Software designers and programmers need to be prepared for unannounced changes in resources, error recovery, raise conditions, etc. Managing these challenges again increases the amount of code that is traditionally not considered to be crucial for understanding the overall architecture of the system.

Maintaining and re-engineering of old software systems is often performed using minimized effort. This, in turn, yields to uncontrolled evolution of the software and increased amount of glue code used. As a result, less and less of the initial architecture is left or it is more difficult to find. By studying the software evolution in various case studies, Lehman *et al.* have noticed that adding new code typically causes increasing complexity, declining comprehensibility, and increasing resistance to future changes [LPRTW97]. Designing a good architecture flexible for changes is a time consuming and challenging task. In addition, predicting future needs for changes is not easy if possible. Even though reusability and good software architecture is often desired, time-to-market usually counts.

Recovering the architecture of a large software system, composed from various components (possibly distributed over the internet) is very difficult. Maintenance, re-engineering, and reuse processes are often task-oriented. The re-engineer or (re)user typically wants to know *how* to use the architecture and the services provided and *how* to add new features taken advantage of existing code, yet respecting the existing architecture. Thus, documentation for the interfaces of different components is especially important. Enumerating different methods that can be called is, however, insufficient; support for understanding which methods to use and how to achieve a desired functionality using existing services would be desirable. As with any software products, from the user’s point of view, a *User’s Manual* is often the most interesting piece of documentation and easiest to understand and use. Extracting just the different components used and ignoring the glue code does not really provide enough information

to the re-engineer. In many cases, that would only lead to even more glue code since the proper mechanism how to modify and (re)use the software is not understood. After the modification has been implemented, documentation should be provided: what are the implemented parts, what is the functionality implemented, how it was implemented, and how to use the implemented parts later on.

To understand the provided services and functionality, we need to understand what they are, how they have been implemented, and how they work. To achieve this goal, the re-engineer needs to comprehend how and what parts of the software are related to certain kind of run-time behavior. Various tools and methods have been provided for combining static and dynamic analysis during the reverse engineering process [KC99, RD99, SKM01]. Modeling and documenting the different aspects of the software architecture is challenging. In traditional reverse engineering tools, various kinds of directed graphs are used to visualize the software components and their relations. Documenting *how* to use the software, in turn, is more complicated. As done in User's Manuals, example use cases and scenarios, possibly including code fragments, can be used for that purpose.

Traditionally, the focus of architecture recovery and reverse engineering has been on structural models that can be analysed for finding answers to questions starting with *What*. When considering the problem from the maintainer's or re-engineer's point of view, we should pay attention on constructing models that give answers to questions starting with *How*. This way of thinking raises new questions and problems concerning information extraction as well as information visualization and documentation.

## References

- [EG01] A. Egyed, R. Balzer, Unfriendly COTS Integration – Instrumentation and Interfaces for Improved Plugability.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Pattern, *Elements of reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [KC99] R. Kazman, J. Carriere, Playing Detective: Reconstructing Software Architectures from Available Evidence, *Automated Software Engineering*, **6**,2, 1999, pp. 107-138.
- [LPRTW97] M. Lehman, D. Perry, J. Ramil, W. Turski and P. Wernick, Metrics and Laws of Software Evolution – The Nineties View, In *Proc. Metrics 97*, 1997, pp 20-32.
- [RD99] T. Richner, S. Ducasse, Recovering high-level views of object-oriented applications from static and dynamic information, In *Proc. Of ICSM'99*, 1999, pp. 13-22.
- [SKM01] T. Systä, K. Koskimies, and H. Müller, Shimba - An Environment for Reverse Engineering Java Software Systems, *Software Practice & Experience*, **31**, 4, 2001, pp. 371-394.