

The Role of Ontology in Software Architecture

A Position Paper for OOPSLA Workshop on How to Use Ontologies and Modularization to Explicitly Describe the Concept Model of a Software Systems Architecture

Juha Savolainen

Nokia Research Center
Juha.Savolainen@nokia.com

1. Introduction

Having conceptual models of software is nothing new. Software design deals with making abstractions of a machine to be built. These descriptions show the problem domain (the environment) and the solution domain (the machine). Both of these domains have entities that manifest different concepts. The collection of these concepts and their connections create the language of the application.

These domains have also reoccurring forms that exhibit some characteristics of the domain. These proven solutions, in the context, are called patterns. The patterns are typically reusable across different domains. They can be adapted to the conceptual model of the particular application. The most commonly known form of the patterns is design patterns (Gamma et.al. 1994). Now also patterns for the problem domain have been proposed to better understand, analyse and classify the problem (Jackson 2001).

There are entities that describe the business rules in the problem domain. These types of rules typically deal with high-level concepts and their allowed collaborations. But in a software project, also much more detailed descriptions are produced. The same project may have a database schema that records the session data for each client that accesses the data storage. Clearly this information is also a part of the overall vocabulary of the system. A typical developer may actually interact with only a small part of the high-level business model but almost all of the concepts of the underlying technological solution concepts.

The complete ontology of an application is (1.) the collection of problem and solution domain concepts. There are (2.) domain independent and clearly domain specific concepts. Different concepts describe (3.) vastly different levels of detail. We argue that an effective ontology should cover all these aspects by allowing all the important concepts to be represented.

2. Modularisation and ontology

Modularisation implies that the software is divided into separate parts that can be treated individually (IEEE 1990). Typical criteria that is used for the modularisation process is cohesion and coupling (Yourdon 1989). Cohesion tries to maximise the logical similarity of the entities that belong to the same module. Typical object-oriented approach would then be to collect similar objects, which somehow collaborate on the same functionality, to provide the services that the module is supposed to give. Object-oriented approaches seem to share a lot same ideas of a common vocabulary with ontology-based approach. It seems to be a good

guess that for each (reasonable large) module ontology could be created. The fact that similar objects are collected together implies that they describe similar concepts, which in turn create a conceptual model of terms. The collection of these ontologies would then create the common ontology for the application as a whole.

Modularisation by clustering classes provides quite limited support for separating concerns. Current research in the aspect-oriented technologies may help to reduce the dependence on mapping the high-level concepts to the class structure. Having aspects that crosscut the class structure provides new tools to describe concepts and implementation that are otherwise cluttered throughout the code base (AspectJ 2003). This has potential to improve our abilities to connect high-level entities to the source code.

Traceability research in this context means trying to connect the ontology to the source code. However, tracing design decisions to source code is difficult. The impact of high-level decisions tends to disperse into vast number of code lines. Looking a simple pointer assignment in the C code does not declare the reason why it takes place. It's a hard to say whether this code line increases the performance by replacing the copying of large data structure or whether it is there to provide flexibility for future extensions.

In fact, the presence of the indirection in form of the pointer it implies a certain performance penalty. The fully understand the contribution of one line of code one must, in theory, understand all dependent lines of code lines, alternative design decisions that could have been taken and the other decisions done before writing the code line in question. Recording the decisions done during the software design and implementation can greatly help this process (Savolainen and Kuusela 2002).

3. Architecture and ontology

Architecture is often described using viewpoints. Each viewpoint describes the system from a unique perspective (IEEE 2000, Hillard 1999). Viewpoints can their own concepts and possibly totally different modelling language (Lorentsen, Tuovinen and Xu 2001). Thus each viewpoint potentially forms a separate ontology. The practise needs to guarantee that the set of viewpoints is consistent and those views describe a software system which is realizable based on these partial descriptions. This is a similar problem to connecting two or more ontologies. We must be able to map separate model into each other.

Just like the semantic web seems to go towards having collections of ontologies the architecture seems to have its own set of possible concerns that may have ontology of its own. One possible approach could be having a number of closely linked ontologies. Already techniques exist that could make this happen in practise. Model-driven architecture (MDA) and UML profiles provide tools to manage set semi-independent collections of concepts and terms. Each of the separate ontologies could be maintained as its own UML profile. The collection of those profiles would then create the complete ontology for the particular application. Additionally, MDA concepts could be used to separate platform independent and dependent parts, where the "platform" refers to the aspects that we want to manage. Using UML profiles and MDA for creating ontology allows creating a hierarchical model structure promoting independent model reuse and incremental introduction of more specific concepts.

4. References

- AspectJ (2003): <http://eclipse.org/aspectj/>, accessed August 2003.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1994): *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley.
- Hillard, R., (1999): Views and Viewpoints in Software Systems Architecture, a position paper for the First IFIP Working Conference on Software Architecture (WICSA1).
- IEEE (1990): *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990, IEEE.
- IEEE (2000): *Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Std 1471-2000, IEEE.
- Jackson, M., (2001): *Problem Frames – Analyzing and structuring software development problems*, Addison-Wesley.
- Loretsen, L., A-P. Tuovinen, and J. Xu (2001): Experiences in modelling feature interactions with colored petri nets. Tibor Gyimothy, editor, *In the Proceedings of Seventh Symposium on Programming Languages and Software Tools SPLST 2001*, pp. 221–230.
- Savolainen, J., and J. Kuusela (2002): Framework for Goal Driven System Design, *in the Proceedings of COMPSAC 2002*, 749-756.
- Yourdon, E., (1989): *Modern Structured Analysis*, Prentice Hall.