Software and Systems Modeling (2024) 23:525–526
https://doi.org/10.1007/s10270-024-01190-0

EDITORIAL

Model modularity for reuse, libraries and composition: symbol management is key

Benoit Combemale[1] · **Jeff Gray**[2] · **Bernhard Rumpe**[3]

Published online: 27 June 2024

It is insightful to observe the similarities between the ways that programs are decomposed into sets of individual and reusable artifacts and the corresponding ways that models are (or should be?) defined. Modularity and encapsulation were introduced in the earlier work on Algebraic Datatypes and then Parnas pointed out the important mechanism of modularity with his famous paper about "On the Criteria to be Used in Decomposing Systems into Modules." Dijkstra also discussed modularity by popularizing the term "Separation of Concerns." Most modern programming languages provide mechanisms for modularity and developers use the concepts of class, module, and assembly in their general programming repertoire. It is a core principle that the internal realization of implementation detail is secretly encapsulated and can only be accessed by an explicitly defined and exported interface. This interface is defined by definitive programming elements, which are given a human-readable name, such as class name "Person" or method name "getAge()".

Modularity has several benefits, according to Parnas, including:

- When the underlying secret is changed, an implementation can be exchanged and evolved without the need to modify the dependent code because the defined interface remains untouched. This supports software changeability while minimizing global impacts across the code base.
- Code can be developed independently within the same project, which supports parallel development by a team of software engineers. This also supports the integration and reuse of external and open-source libraries. Since McIlroy's paper on reusability in the late 1960s, the general reusability of library code and frameworks has greatly increased programmer efficiency.
- Modularity also helps with the comprehensibility of the program code by allowing each module to be reasoned about locally without a deep need to understand all of the inner details of other parts of a project.

On the contrary, modern modeling languages and their tools do not natively provide a deep collection of reusable libraries. One key problem is the lack of well-understood, encapsulating interfaces of the defined models. So far models have often been used to describe the interface of an underlying component, but the concept that a model itself has an interface and an encapsulated "body" is still unfamiliar. Only a few works from the community promote this idea, such as the concept of symbol tables for models, potentially aggregated to a model type or requirement model. This idea also has been further explored with different levels of interfaces, such as the interfaces for variability, customization, and use (VCU).

Moreover, the availability of interfaces on models means that there must exist an efficient management of symbols. Regarding symbols, we mean named elements that modelers define inside a model that are allowed to be referenced from outside, using the defined name and signature (as provided by symbol tables, model types, or requirement models). This includes classes, states, activities, pins, ports, methods, attributes, variables, and many more potential kinds of symbols.

Symbols are a well elaborated concept in programming languages (e.g., symbol tables explicitly store symbols in separate or joint artifacts). But symbol management is often absent in modeling languages and thus not very well accommodated in most tools. Most modern UML and SysML tools do not manage symbols explicitly, but store an integrated large model, where elements are directly connected, even though models provide explicitly defined names for their

✉ Bernhard Rumpe
bernhard.rumpe@sosym.org

Benoit Combemale
benoit.combemale@sosym.org

Jeff Gray
jeff.gray@sosym.org

[1] University of Rennes, Rennes, France

[2] University of Alabama, Tuscaloosa, AL, USA

[3] RWTH Aachen University, Aachen, Germany

elements. Integrated storage is a successful database management technology, but it is not clear how well the concept fits modeling languages, where decoupling, encapsulation, parallel development, and independent reuse play a major role. To our knowledge, there are very few programming languages that use database technology to store their artifacts (and developers often find such an approach unsatisfying). Instead, source code is stored in individual artifacts and only connected through explicit symbols, typically imported from other artifacts. Modeling languages are also complicated data structures, and thus database technologies are not necessarily the optimal solution.

As an alternative, an approach where artifacts are individually stored, managed by a version control system and connected through explicit use of symbols in the interfaces of the models, could be a more promising approach to support the reuse of individual models and libraries of well-defined models. Such an alternative approach, however, may comprise two relevant aspects:

1. The notion of a model's "interface" has to be made more explicit in the modeling languages. Although not completely independent, a model's interface is not to be confused with the interface of the system components that are described by the model. There are (sometimes subtle) differences.
2. The notion of "symbol" is the core constituent of what makes an interface. Symbols have a name and need an explicit type system (relatively well-known in programming languages). As exemplified above, there are language-specific kinds of symbols that are not present in programming languages. For example, a UML class diagram defines multiple kinds of symbols, namely (a) class names (including interfaces), (b) attribute names, (c) method names, (d) stereotype names, (e) association names, (f) role names, (g) enum constants, (h) package symbols, and finally (i) the diagram name itself. Some of those names are directly mapped to the same symbol in the program code, while other symbols of the abstract model map to other symbols in the implementation (e.g., an attribute's get/set access methods). Other kinds of symbols also have to be mapped, because there is no corresponding programming symbol (e.g., association, role, and diagram names).

The mapping is also not unique. For example, a state symbol in a StateChart may become (a) an enumeration constant , (b) a class in the state pattern, or (c) not visible at all, because the decision was to encapsulate states within the modeled software component. Symbols also contain additional information, such as method signature, typing, and visibility. To achieve modularity and compositionality in a practically useful form, a better theory on symbols, symbol management, and the exchange of symbols through models seems to be necessary. The software and systems modeling community has undertaken considerable steps in this direction, and we know it is not easy, but we think that smart interface management is a highly relevant and widely undervalued technology in modeling language design.

Finally, our plea to the SoSyM community is to look more deeply into the mechanisms of composing respectively decomposing models into individual, reusable artifacts with a crisp boundary and the possibility to explicitly access exposed parts of the models through interfaces. Our modeling approaches need interfaces that are explicitly constituted by named symbols of various appropriate kinds, such that models can be composed and integrated, but also packaged and evolved individually as part of libraries. We strongly believe this will help to make modeling more successful.

We are interested in receiving submissions and comments from authors who have been able to expand concepts of modularity in languages such as SysML, UML and customized domain-specific languages.

Funding Open Access funding enabled and organized by Projekt DEAL.