Interfaces in Modular Software Systems: Some Research Questions

LEIF BONORDEN
UNIVERSITÄT HAMBURG
bonorden@informatik.uni-hamburg.de

Abstract

Modularity of software systems is well-known and supported by various theories. Interfaces and interactions between such software modules are differently seen and treated from different points of view.

This article briefly surveys semi-formal models, formal specifications *and* technical implementations, *and introduces corresponding research questions regarding the compatibility of these perspectives with each other and their role in the software development process.*

1 Modularity in Software Systems

The division of software systems into *components* or *modules* to deal with complexity has been observed for decades [7]. In particular, object-oriented software development is an instrument to structure software into components – classes and objects –, but nowadays the term *module* is used to describe larger elements of software. In 1988, Bertrand Meyer presented five criteria for such modularity in a software development process [5]:

- **Decomposability.** Division into less complex structures that can be examined and developed separately.
- **Composability.** Modules can be combined with others in different settings, in particular outside the original setting.
- **Understandability.** A human can understand each module without knowing about others or only a few others.
- **Continuity.** Small architectural changes in one module affect no or only some other modules.
- **Protection.** Problems at run-time in one module affect no or only some other modules.

The five criteria are used together with five rules and five principles to characterise modular software design. Following this approach, various ideas of modularization have emerged – either technical, e.g. service-oriented architecture (since 1998)[8], and micro-services (since 2014)[2], or organizational, e.g. domain-driven design (since 2004)[3].

2 Interfaces for Software Modules

In general, interfaces are considered to be the part of a module that is common to – or commonly used by – the module implementation and its environment. A special case, which will not be further considered here, are user interfaces. Thus, in the following the environment also consists of other software modules and systems.

Meyer's five criteria for modularity yield three rules for modular software design that deal with interfaces [5]:

- **Few Interfaces.** For each module, communication is limited to as few other modules as possible.
- **Small Interfaces.** The communicated information between modules is as small as possible.
- **Explicit Interfaces.** The communication of two modules should be done explicitly and comprehensibly.

Based on these rules and similar results, software architects design systems, modules and interfaces. Besides *module interfaces*, the more general subject of *module interaction* may also be considered.

2.1 A Semi-formal Modelling Perspective

Using the Unified Modeling Language (UML), interfaces may be depicted in class diagrams or in component diagrams. In class diagrams, interfaces may be explicitly marked with the stereotype *«interface»* and are otherwise identical to classes. In component diagrams, the interaction between components or modules is highlighted by explicit notation of interfaces. Due to the *semi*-formal notation of UML, the semantics of these notation may differ significantly.

2.2 Formal Perspectives

Mathematical foundations have been developed in order to reason abstractly about interfaces. Among the areas of active research on such formal instruments are the following.

Modal Specification Modal automata specifications distinguish two types of transitions: *must* and *may* – formally a tupel $S = (A, \rightarrow_{must}, \rightarrow_{may})$ with an alphabet A and partial functions

$$\rightarrow_{must}, \rightarrow_{may} \colon A^* \mapsto 2^A$$

with the condition of

$$\rightarrow_{must}(a) \subseteq \rightarrow_{may}(a)$$

for each $a \in A$. A system implementing such specifications has to include all *must* transitions, but does not need to include all *may* transitions [4].

Interface Automata The transitions of an interface automaton are labeld as *input* or *output* – formally a tuple $\mathcal{P} = (X, x_0, A, \rightarrow)$ with state set X, an initial state x_0, an alphabet A and a transition relation

$$\rightarrow \quad \subseteq \quad X \times A \times X.$$

The theory of these automata is game-based, i.e. two players represent the environment (*Input*) and the module (*Output*), respectively [1].

Design by Contract In contrast to modal specifications and interface automata, contract theory focuses on *pre* and *post conditions* for interactions. A *contract* is defined as a set of variables (*input variables* and *output variables*) together with their *types* as well as *assumptions* and *guarantees*, which lay out the responsibilities of the environment and the module, respectively: While the environment needs to ensure the *assumptions*, it may expect the *guarantees*. On the other hand, the module may expect the *assumptions*, but needs to realize the *guarantees* [6].

2.3 Implementation Perspectives

On the programming and technical level, there are a variety of different perspectives on module interfaces and module interaction. Ranging from usage of programming language features and middleware to the choice of transmission protocols, module communication needs to be considered in various development activities.

3 Further Considerations on Interfaces

Apart from different perspectives on interfaces and module communication, additional considerations are needed:

- Linkage between modules may be determined **statically** – favoring analysis – or **dynamically** – increasing flexibility.
- Provided interfaces and communication between modules need to be considered from a **security perspective** as they offer usual attack vectors.
- Design and implementation of interfaces and communication need to be **traceable** across development phases and artifacts.
- Interfaces need to be considered together with, but also **separated from their implementations** in order to allow future changes to the implementation or the system.

4 Research Questions

The intended research considers software construction processes with a special highlight on interfaces. In this setting, the following research questions arise. To the best of the author's knowledge, they have not been sufficiently answered in this context, but the results that have already been achieved, for instance in formal specification, need to be taken into account.

How can the different perspectives on interfaces and module interaction be considered jointly?

- To what extend are the several formal views on interfaces compatible?
- Can formal specifications of interfaces and module interaction be profitably connected to semi-formal models?
- How do programming and technical details result from (semi-)formal models of module interfaces and module interaction? How can this be supported in practice?

How can theories with formal basis further advance general (not necessarily formal) software development with respect to module interfaces and module interaction?

- Can certain flaws in the design of a system be detected through formally based analysis?
- How can formal theories aid the software development in checking whether an implementation satisfies its specification?

Are the additional considerations describable in the different perspectives?

- How can security aspects be integrated into (semi-)formal models of module interfaces and module interaction?
- How can models and implementations of module interfaces and module interaction be linked?
- How can architectural knowledge be made available through models and implementation?

References

[1] Luca de Alfaro, Thomas A. Henzinger. *Interface Automata*. ACM SIGSOFT Software Engineering Notes, vol. 26, no. 5, pp. 109–120, 2001.

[2] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*. In: Manuel Mazzara, Bertrand Meyer (eds). *Present and Ulterior Software Engineering*. Cham: Springer, 2017, pp. 195–216.

[3] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2004.

[4] Kim G. Larsen. *Modal Specifications* International Conference on Computer Aided Verification. Berlin, Heidelberg: Springer. 1989.

[5] Bertrand Meyer. *Object-oriented software construction*. New York: Prentice Hall, 1988.

[6] Bertrand Meyer. *Applying "Design by Contract"*. Computer, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[7] David L. Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules*. Commun. ACM, vol. 15, no. 12, Dec. 1972, pp. 1053–1058.

[8] Mohammad H. Valipour, Bavar Amirzafari, Khashayar N. Maleki and Negin Daneshpour. *A brief survey of software architecture concepts and service oriented architecture*. 2009 2nd IEEE International Conference on Computer Science and Information Technology, Beijing, pp. 34–38, 2009.