Characterizing Architecture as Abstractions over the Software Domain*

Eyðun Eli Jacobsen **Bent Bruun Kristensen** **Palle Nowack**

The Maersk Mc-Kinney Moller Institute for Production Technology
Odense University, DK 5230 Odense M, Denmark
e-mail: {jacobsen, bbkristensen, nowack}@mip.ou.dk

Abstract

The design of software architecture is seen as abstraction over the software domain. In this perspective architecture is characterized according to the process, the notation and the principles involved. The characterization is seen from a research and a practical point of view.

1 Introduction

In [6], [1] and [5] software architecture is characterized—or even defined—in various different ways. In every software system, some architecture is present. The problem is that the architecture is only implicitly available. The architecture did exist for the developers during the design phase, but because it was not expressed explicitly, the architecture either has been lost, or only has been reproduced to some extent in the software documentation.

None of the above mentioned definitions of software architecture are used in this article. Our overall thesis is that software architecture is abstractions over the software domain. The underlying model of this thesis is presented in [3]. The immediate consequences of this thesis include that

(1) The software must have been produced or at least be anticipated, i.e. the software domain must exist in some form. The software domain is given by descriptions—the focus during the architecture design is on these descriptions. The result of the design is concrete/abstract *descriptions* (notation/diagrams) in addition to the software domain. But there is no given sequence—the architecture may be created as a *prescription* of some anticipated software.

(2) The architecture appears to be *redundant* descriptions. Because the software domain is available the architecture design seems to be an additional and even redundant description. The challenge is to clarify why it is necessary to elaborate on the software description, to make an understanding / a mental model explicit, and to create abstractions over the software domain.

(3) Architecture design requires abstraction. Abstraction implies, in addition to organization and delimitation, the formation of concepts at another abstraction level together with the use of the concepts at that level. We abstract from irrelevant details and concrete matters, given selected perspectives. But we have to find, introduce, and motivate the abstractions (and the perspectives). Enough experience in architecture design and insight in the given application is necessary to be able to come up with suggestions for *the right abstractions*, and to decide which are the correct and promising ones.

(4) The nature of abstractions over the software domain is still partially unexplored and unfamiliar. In object-oriented modeling our abstractions are concrete in the sense that the abstract matters are related to the problem and the usage domains. Abstractions over the software domain are different—probably more abstract and unreal. The challenge is to clarify the *essence of software abstractions*, and specify the quality measure for the abstractions.

(5) The architecture design process must be covered by the *software development methodologies*. The architecture is only superficially introduced, usually only mentally, despite the fact that lack of the support of this in the methodologies. Architecture design *is* difficult, and because architecture design is not covered by the methodologies the general situation is that we have little experience, no good examples or prototypical cases. The methodologies do not promote (the acceptance of) reuse of design—reuse of code is still the rule.

Article Organization. In section 2 we include a model of the design phase in the software development process. We also include a model of the abstraction processes involved during the software development process. The section appears as an extract from [3]. In section 3 we characterize the architecture design in the perspective as abstractions over the software domain—organized according to the process, the notation and the principles of the design phase. We briefly comment on the current divergences between the theoretical models and practical use of software architecture.

*This research was supported in part by Danish National Centre for IT Research, Project No. 74.

2 Design and Abstraction

Model of the Design Phase. From the analysis phase the *problem domain model* and the *usage domain model* are available, cf. the general model for object-oriented software development in [3]. In the design phase the developer uses the usage domain model to refine the problem domain model into a system model, and introduces an architecture model. These parts are related, but an explicit and abstract description of the architecture model is essential. The architecture model gives an supplementary perspective, is typically a partial model, is abstract in the sense generalized and generic (parameterization), and describes the logical organization of the system model and the (logical) execution platform. It describes which parts are distributed, which are concurrent, which are to be persistent etc. to enable the developer to transform the problem domain model into a a model that can support the usage of the system, not only the understanding of it.

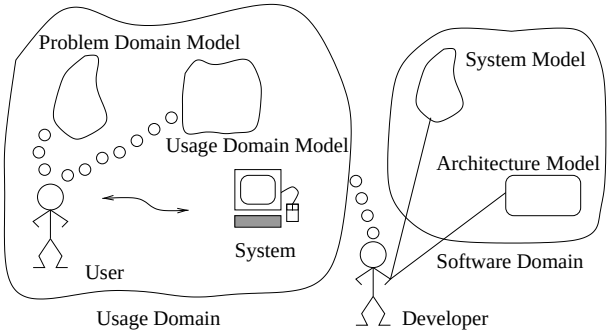

Figure 1: Domains and Models of the Design Phase.

Figure 1 (from [3]) illustrates the model of the design phase with the usage domain and the software domain. The *system model* forms the developer's conception of the integration of the problem and usage domain models at an abstract level. It is a refined, transformed and enriched problem domain model. The system model also supports the usage of the system, and not only the conception of the problem domain. The *architecture model* forms the developer's conception of the architecture of the system, i.e. the overall structures and their relations and interactions. It is an abstract model over over the system model. The model focus on (the organization of) the structure and interaction embedded in the system model. The purpose is to understand the system model given a structure/interaction perspective on that model, to allow us to reason about and expose the support for the non-functional requirements, and to map the system model onto the logical platform. The *software domain* includes descriptions, partial or complete, of some software. Various design notations are used.

The result of the design phase is not a model of some **existing** domain similar to the problem domain or the usage domain. Rather, it is a unique design, something constructed and created by the developer based on his skills and knowledge, and from the problem and usage domain models. After its construction the system model and the architecture model both works as models in addition to the problem and usage domain models, but from a specific design perspective.

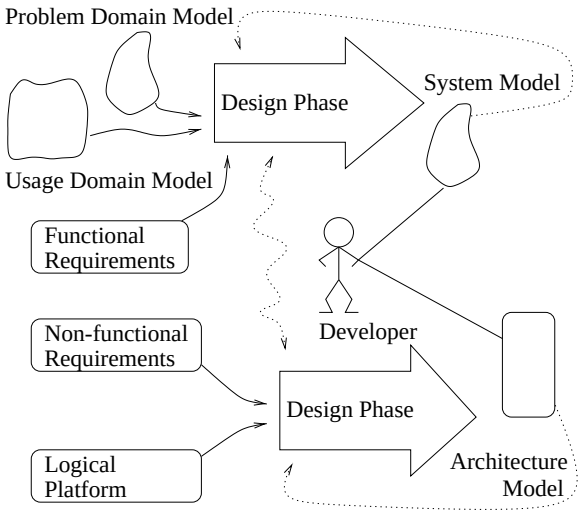

Figure 2: The Design Process.

Figure 2 (from [3]) illustrates the design process, its inputs in the form of models and requirements, and its resulting models. The dotted relation between the design arrows in Figure 2 symbolizes that the two design parts have influence on each other. On the one hand the problem domain model and the system model are the foundation for creating the architecture. The usage model is used during the design for controlling the transformation of the problem domain model into the system model. The usage model, together with the functional requirements, supply the information for adding and distributing the functionality to the problem domain model to the extent that this information is not already there. The architecture is a constructive solution for how the non-functional requirements and the organization of the logical platform can be combined. The architecture model created by the design arrow at the bottom can have crucial effect on the actual transformation at the top arrow. The *functional requirements* is a set of requirements concerning the capabilities of a system—what the system should be able to support. The *non-functional requirements* is a set of requirements concerning the non-functional qualities of a system—for example usability, security, efficiency, correctness, reliability, maintainability, testability, flexibility, understandability, reusability, porta-

bility, adaptability. The *logical platform* is a description of the platform on which the system is going to be executed but at a logical level—i.e. requirements in terms of user interface, distribution, persistence and concurrency.

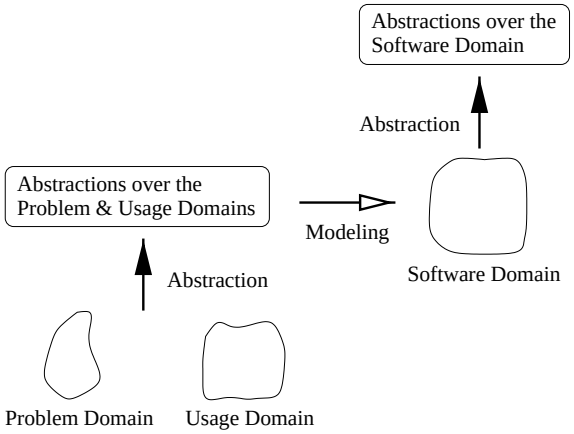

Figure 3: Abstractions & Models.

Abstraction. Figure 3 (from [3]) illustrates the difference and relation between two kinds of abstraction involved during the software development process. During analysis we form abstractions over the problem domain and the usage domain—the abstractions model concepts and phenomena in these domains. We need an understanding of these domains and some notation for expressing the abstraction. During design (and implementation) we form abstractions over the software domain—such abstractions do not model anything from the problem and usage domains (at least only indirectly). We need a notation for describing a model of the software—we focus on the software and we form abstractions over the software. Such abstractions are (to a certain extent) dependent of the notation used for the software [5], but application domain independent.

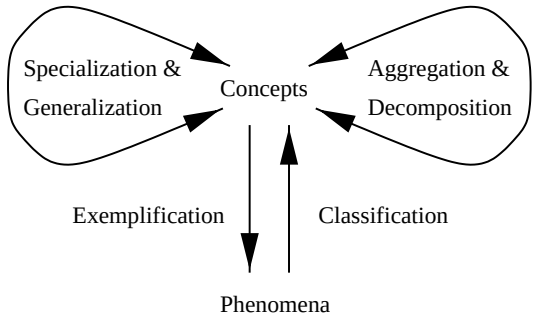

Figure 4: Abstraction Processes.

Figure 4 (from [3]) illustrates the abstraction processes in conceptual modeling. Conceptual modeling can be seen as the underlying theoretical understanding of object-oriented programming although existing object-oriented notation only in a limited way supports this theory. But the processes are fundamental abstraction processes. An object/class is seen as a model of a phenomenon/concept—to give the class of an object corresponds to classification—to give an object of a class to exemplification. To let a (specialized) class inherit from a (general) class corresponds to form a specialization of that (general) class. To describe a (whole) class by the use of references to (part) objects corresponds to form an aggregation of the (part) objects.

Abstractions can support both the organization of the software (with respect to the development platform) and with respect to the (logical or physical) execution platform. The latter is described by the architecture model. The former, which is covered by for example module descriptions and is important for software version and configuration control, is not considered part of the architecture of the system and therefore not discussed further. Architectural abstractions can be supported by means of patterns [2], frameworks [4] and architecture notation/languages (AL's) [6]. Both pattern and framework technologies are powerful means of capturing abstractions over the software domain. Still the notation used for the representation of these is usual object-oriented notation—we simulate the architectural description. Only preliminary elements of specific architectural notation/languages are available, and the understanding of the necessary expressive power of such a notation/languages is mainly captured through architectural styles (pattern-like abstractions at the architecture level) [6] and [1].

3 Characterization

In the previous sections the design of software architecture is presented as abstraction over the software domain. Given this perspective we characterize architecture according to the process, the notation and the principles of the design phase for the architecture. The characterization is organized as a number of theses. The theses are based on the previous section, and can be seen as representing a research point of view of architecture design. This theoretical characterization is complemented by remarks from a practical point of view. This view represents a combination of current practice, expectations, what is actually possible, etc.

Process. The process of the architecture design is characterized as follows:

Thesis: Architecture design is innovative construction, not just modeling or transformation. Design of architecture is not just modeling: There are no domains to model

(like in the analysis phase, where the developer can take the starting point in the problem and usage domains). Design of architecture is not just transformation: There is no existing model available (like in implementation where the model resulting form the design has to be transformed into another model at the programming language level).

In practice the innovative processes give problems both because they are difficult and expensive, and also because they are hard to control and administrate.

Thesis: Architecture design implies abstraction, not just organization. By organization we mean the process of arranging and relating subparts of wholes. This resembles aggregation and association to a large extent. Abstraction includes organization—in the sense that substance has to be identified and delimited. But abstraction also involves classification and generalization—a given chunk is classified according to various existing or new concepts—the concept is related to other concepts. In addition the abstraction is thereafter treated at its abstract level.

In practice it is straightforward to organize material. During that process we get some knowledge about the material. But by organization only we more or less just view information at the given level. To obtain important knowledge, we have to abstract. Abstraction is much more demanding than organization.

Notation. The notation applied during the architecture design is characterized as follows:

Thesis: Architecture notation can be informal. Any notation support the thinking of—and description of—solutions and models, but it also introduces limitations. To be innovative we need expressive freedom, for example by the use of informal notation.

In practice we seem to prefer complete/formal specifications, and we need to include all details also concerning the architecture. Only with a formal notation the power of reverse engineering can possibly be obtained.

Thesis: Architecture notation can be several notations. During the innovation of a model several notations can be applied simultaneously, possibly to various partial models such that each of these models can be expressed in a special notation (for example type of diagrams). In addition, the models can be created from different perspectives—often overlapping and containing redundant information. Redundancy supports the understanding.

In practice several notations and corresponding models introduce an inconsistency problem. It is much more convenient with one unifying (but insufficient) notation, and it is much more straightforward to handle one unifying model. By different perspectives we usually introduce conflicting models to be resolved, and ultimately we have to choose among the perspectives, or go through a complicated process to integrate and unify the perspectives.

Principles. The principles underlying the architecture design are characterized as follows:

Thesis: Architecture must be developed explicitly. The elements included in the methodologies for architecture design are still preliminary. The result of the design is not described explicitly. Our understanding of abstraction over software is still insufficient. The evolution process where architectural abstractions and abstraction mechanisms of architectural languages are developed further by mutual influence has not really started—it has started with architectural styles and patterns, but the languages are still not available.

In practice we mostly describe the architecture implicitly as an integrated part of the other models. When it is actually explicitly described/illustrated, it is usually done in object-oriented languages, as it is the case for design patterns and frameworks. The architectural abstractions are typically related to the problem and the usage domains. The abstract world over the software domain is not explored in praxis.

Thesis: Architecture design must be supported by tools. Principles are usually (at least partially) supported by tools. Tools rely on notation. Because there is no unifying notation and the notation is partially informal, etc. the tools for architecture design are not very powerful—they are more like simple drawing tools.

In practice the tools constitute the principles (and define the praxis) in many organizations. Tools are important for the process, the notation and the documentation standards.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture: A System of Patterns. Wiley & Sons, 1996.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[3] E. E. Jacobsen, B. B. Kristensen, P. Nowack: Models, Domains and Abstraction in Software Development. Proceedings of International Conference on Technology of Object-Oriented Languages and Systems, 1998.

[4] R. E. Johnson, B. Foote: Designing Reusable Classes. Journal of Object-Oriented Programming, 1988.

[5] B. B. Kristensen. Architectural Abstractions and Language Mechanisms. Proceedings of the Asia Pacific Software Engineering Conference, 1996.

[6] M. Shaw, D. Garlan: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.