Software Architectures—Quality for Software Design

Prof. Jukka Paakki

Department of Computer Science, University of Helsinki

P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland

E-mail: paakki@cs.helsinki.fi

Abstract

Software architectures are the cornerstone of software design and, consequently, one of the main factors of software quality. We present some of the central issues in software architectures, concentrating especially on those that are currently subject to intensive scientific research and industrial development.

1 Introduction

A typical software development process consists of several phases, executed either sequentially or iteratively depending on the underlying general development model. Whatever the model, the actual coding (or implementation) phase in systematic software development is always preceded by a phase commonly called *software design*. In the design phase, the system under development is studied at an abstract level, so as to find the most appropriate technical software solutions to what is required of the system. Software design, serving as a bridge from informal and imprecise customer requirements to formal and precise program code, is the most central development phase with respect to the quality of the resulting system.

The design phase can be further refined into subphases in several ways. Usually at least two subphases can be distinguished: *architecture design* and *module design*. In architecture design, the whole system (or its software part) is addressed at the level of subsystems, modules and their interrelationships, while in module design the emphasis is on internal data structures and algorithms to be applied for the implementation of the modules. The transition from the level of architecture design to the level of module design may entail several iterations of an increasing amount of detail; for instance, there might be an intermediate phase called *subsystem design* where closely related groups of modules are identified as logical entities.

In recent years, the software architecting phase has been steadily growing in importance [13]. Since a software architecture captures the early design decisions, it serves as an important communication, reasoning, analysis, and evolution tool for systems. Furthermore, the emerging special architectural design methods, languages and tools have made *design reuse* a reality even in practice. Finally, as observed by several studies, a good (software) architecture is one of the major cornerstones of the quality of a (software) system.

In this paper we address the content and the main research areas of software architecting. We start by discussing the notion of software architecture in some more detail in Section 2. Different views of software architecture are presented in Section 3, with the so-called "4 + 1 model" as the case example. In Section 4 different architectural styles are introduced. Dedicated architecture description languages are characterized in Section 5. In Section 6, some popular object-oriented tools for architectural engineering of software are introduced. Reverse-architecting issues are discussed in Section 7. The use of software architectures in product lines and product families is discussed in Section 8. Finally, future trends are explored in Section 9.

2 A Definition

The term "software architecture" was coined as soon as in the 1970's, at that time meaning roughly the same as a "user interface", that is, those properties of the system that could be visually perceived and analyzed by the user (see, e.g., [2]). Since then, both software architectures and user

interfaces have evolved into disciplines of their own, and especially the meaning of a "software architecture" has been more or less reversed from its original content: in contrast to the external viewpoint of the 1970's, the *internal* (invisible) properties of a system are now being emphasized.

A myriad of definitions for "software architecture" have been given, ranging from intuitive explanations to formal definitions. The most common suggestions are unified in the following characterization where the key terms are emphasized:

> *A software architecture is an* ***abstract*** *description of the components of a software system, the externally visible properties of those components, and the relationships among them—the* ***structure*** *of the software. The structure is typically specified in different* ***views*** *to show the relevant functional and non-functional properties of the system. A software architecture is a* ***technical*** *artifact: the result of the software* ***design*** *activity and a realization of the design decisions. It must be possible to* ***evaluate*** *an architecture, so it must be given in a (semi-)formal* ***notation****.*

This characterization lists the attributes that are considered essential for software architectures. An architecture is necessarily more abstract (high level) than the actual code of the system, yet precise to the extent that it can be analyzed. This property calls for architecture description *languages* that have a suitable selection of abstractions in their vocabulary. Since the architecture of a complex system is quite a large composite description, it must be possible to study it from several different points of view, each possibly with a description (language) of its own kind. Finally, an architecture is a technical description of the software structure and not just some ad-hoc drawing (even though most of the architecture description languages are graphical).

Since a system can be described in several ways, depending on the applied precision and selected viewpoint, it is actually not sensible to talk about *the* architecture of the system but instead of *an* architecture of the system. On the other hand, each software system has an architecture because every system contains some form of "components" and their "interrelationships"—in the trivial case the entire system can be considered as a "component", and at the other end of the spectrum the

individual program statements can be considered as "components" and their execution order as the "interrelationship".

Even though the definition above emphasizes the structure of the software, an architecture also encompasses the *behavior* of the components as a certain kind of property. This behavior is what allows components to interact with each other. Thus, a software architecture cannot be analyzed just in terms of its static structure, but some degree of the run-time properties must be involved as well.

3 Architectural Views

A software architecture is usually a complex entity. To manage the complexity, it must be possible to study an architecture from different integrated viewpoints. One popular set of views for software architectures is known as the "4 + 1 model" [9].

The model organizes the description of a software architecture into five views, each addressing a specific set of architectural aspects. The design decisions are captured in the four main views which are validated and integrated by the fifth view. The views are the following:

1. The *logical* view describes the system's static software structure, as derived from the problem domain. When using an object-oriented (architecture) modeling method/language, such as UML (Unified Modeling Language) [14], the logical view is expressed through a *class diagram* (perhaps with the assistance of more dynamic *interaction diagrams*, *statechart diagrams*, or *activity diagrams*).

2. The *process* view describes the system's concurrency, distribution and synchronization aspects, in UML with *interaction diagrams* (class diagrams, statechart diagrams, activity diagrams).

3. The *development* view shows the organization of the system in terms of technical facilities of the development environment: subsystems, layers, libraries, modules, files, etc. In UML the development view can be expressed as *component diagrams* (interaction diagrams, statechart diagrams, activity diagrams).

4. The *physical* view describes the mapping of the system onto hardware, telecommunication facilities, sensors, databases, etc., reflecting also its distribution aspects. In UML, *deployment diagrams* (interaction diagrams, statechart diagrams, activity diagrams) are provided for this purpose.

5. Finally, the fifth view, the *scenarios* ties the main views together into externally usable system services and interactions between the main elements of the system. In UML, *use case diagrams* (interaction diagrams, statechart diagrams, activity diagrams) serve the needs of the scenario view.

As an example, Figure 1 shows the architecture of a conventional language compiler as a UML class diagram (logical view).

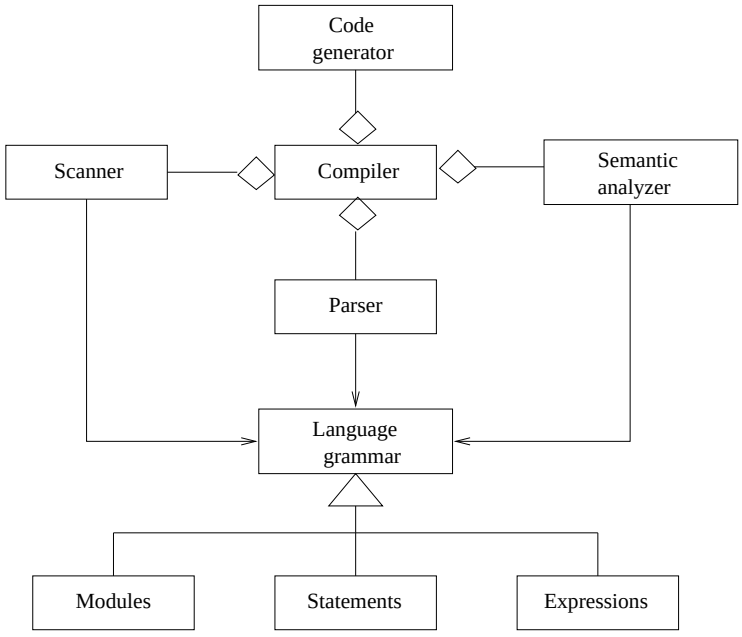

Figure 1. *Logical view of a compiler*

4 Architectural Styles

Software systems in the same application area typically have the same underlying overall architecture. For instance, a compiler always has a "scanner", a "parser", a "semantic analyzer", and a "code generator" as its main components, irrespective of the source language. When capturing the fundamental elements of the common architecture into a general blueprint, something called an *architectural style* is obtained [15]. An architectural style in a sense resembles a programming *paradigm*: a style is not an architecture by itself but instead a set of component patterns and constraints on them so as to provide a skeleton for the actual software architecture.

More precisely, an architectural software style is determined by a set of component types (e.g., data repository, process, procedure) that perform some function at runtime of the system, a topological layout of the components indicating their runtime interrelationships, a set of semantic constraints (e.g., a data repository is not allowed to change its values), and a set of connectors (e.g., subroutine call, data stream, pipe) that mediate communication, coordination, or cooperation among components. Different architectural styles have been classified into categories according to which stylistic features they emphasize. One classification, commonly referred to, is as follows [15, 1]:

- *Data-flow* style, dominated by motion of data through the system. More specialized substyles in this category are "batch sequential", "data-flow network", and "pipes and filters".
- *Call-and-return* style, dominated by order of computation, usually with single thread of control. Specialized substyles are, e.g., "main program / subroutines", "abstract data type", "call-based client-server", and "layered".
- *Interacting process* style, dominated by communication patterns among independent, usually concurrent processes. Specialized substyles: "event system", "communicating processes", "broadcast", "token passing".
- *Data-centered* style, dominated by a complex data store which is manipulated by independent computations. Specialized substyles: "repository", "blackboard".

- *Data-shared* style, dominated by direct sharing of data among components. Specialized substyles: "hypertext", "lightweight processes".
- *Virtual machine* style, characterized by translation of one instruction set into another. Specialized substyles: "interpreter", "compiler".

As an example, Figure 2 illustrates the general style of "layered" software architectures.

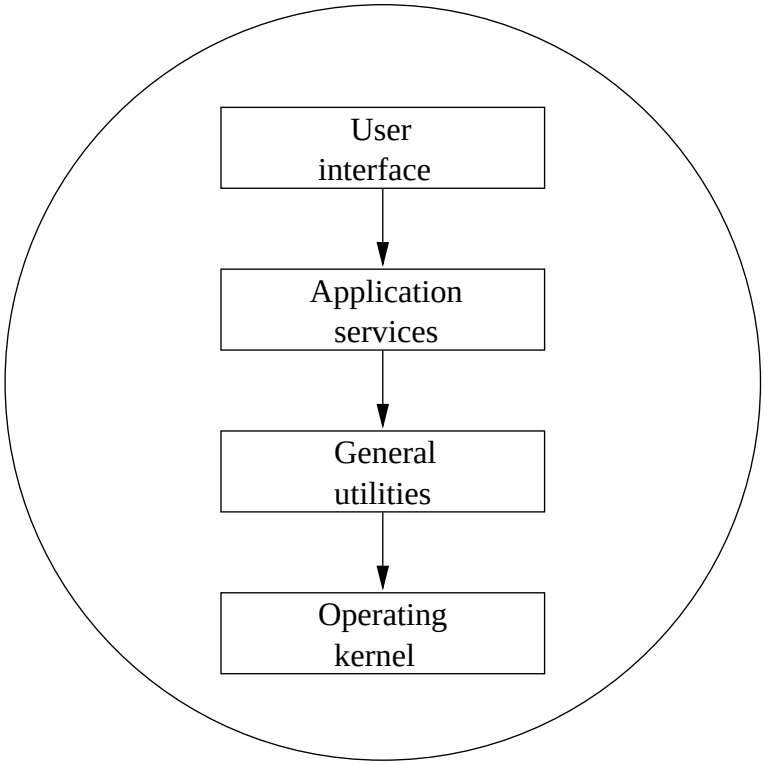

Figure 2. *Layered software architecture*

5 Architecture Description Languages

To describe architectures, one needs languages. Unlike programming languages which need to express computations at the detailed level of algorithms and data structures, architecture description languages must be more high level so as to serve the abstraction requirements of software architecting. Due to this inherent abstraction, architecture description languages tend to be visual (and object-oriented).

To date, architectures have largely been represented as "box-and-line" drawings whose meanings are more or less based on subjective intuition. In large scale software engineering such imprecise drawings are, however, problematic because different people can interpret them in different ways, and because ad-hoc drawings cannot be used for verifying the syntactic or semantic quality of the architecture nor as the basis for further development.

This is why specialized architecture description languages are currently being developed as a technical solution for representing and analyzing software architectures. According to the main elements of architectural styles, an architecture description language must provide facilities for describing the components and connectors of the system, as well as the static constraints and the dynamic interrelationships among them. Moreover, the language must have a precise syntactic and semantic definition to facilitate architectural analysis, or code generation from architecture descriptions.

The issue "dedicated architecture description language *vs.* general modeling language" is controversial. On one hand it sounds useless to develop dedicated languages because the existing modeling languages (especially the rather extensive UML) can well be used even for describing architectures, but on the other hand the linguistic features of the modeling languages might be too general and not designed especially for the purposes of software architecting. For instance, a general modeling language does typically not provide any special support for the description of communication ports and protocols that are absolutely essential when working with the architecture of any conventional distributed system.

Figures 1 and 2 can be seen as architecture descriptions, written in an architecture description language. In the former case the applied language is the general modeling language UML and in the latter case a hypothetical architecture description language.

6 Object-Oriented Architectural Facilities

Software architectures are created by human software designers. The area of software design has a relatively long history in computer science with a large number of theoretically and practically sound technical principles such as modular programming, information hiding, coupling and cohesion. An experienced software designer makes use of these folklore techniques, assisted with her own background knowledge and personal rules of thumb.

Unfortunately, the solutions are often introduced into a software architecture implicitly by mere intuition of the designer, which makes it extremely hard to grasp the design decisions embedded into the architecture. This makes it difficult to understand, maintain and reuse the architecture in the later phases of the software life-cycle.

In recent years, the object-oriented software community has experienced an active *pattern movement*. The most popular concept that has been introduced is that of a *design pattern*, that is, a general and mature solution to a design problem in a context [6]. Design patterns are a solution to the problem of hidden design decisions because they make the decisions explicit: a design pattern applied by a software designer has a particular structure with associated documentation which serve the needs of understanding and maintenance. Furthermore, the design patterns documented in text books and other sources typically support the reuse and extensibility of designs by being based on advanced object-oriented facilities such as interfaces, subclassing, and polymorphism.

There are different levels of patterns. In [4], the patterns are classified into *architectural patterns* (large-scale, fundamental structural organization scheme for complete software systems), *design patterns* (medium-scale, general structures over a small number of interacting objects to solve a design problem), and *idioms* (small-scale, language specific programming tricks to solve an implementation problem). (Design) patterns have been abstracted further into canonical forms, "meta-patterns", that capture their common characteristics [12]. The counterpart of "good" design patterns, "bad" *anti-patterns*, have also been documented as wrong solutions to common (design) problems bearing negative quality consequences [3].

A well-founded software architecture typically contains several instances of different kinds of patterns. In the terminology of [4], the highest

level of an architecture might be based on some (domain-specific) architectural pattern, which might apply a number of design patterns, which finally might take the form of implementation idioms at the most detailed level. Typically some of the patterns overlap in which case a class, object, or component has a certain role in several different patterns. The term *pattern language* is used for such a set of interrelated patterns whose systematic application in a given order yields a complete software architecture [5].

As the final object-oriented contribution to software architecture, an *application framework* is a partially complete software system that is intended to be modified and extended [10]. Unlike the aforementioned patterns that are design-level abstract entities, a framework is made of concrete program code. However, a framework is not a complete implementation of a system but instead an incomplete code skeleton which must be extended in order to obtain an actual application. A mature framework provides a maximal degree of (code) reuse since it can be applied as the basis of several similar systems in its application area instead of hard-coding a single system only.

Frameworks and patterns are closely related. Since (design) patterns are mature and extensible solutions to software design problems, they provide a sound basis for constructing the (extensible) architecture of a framework. Hence, the patterns applied in a framework also serve as its documentation [8]. Moreover, often the extension/instantiation points of a framework (so-called *hot spots*) are nicely captured by the embedded design patterns.

Since (design) patterns and frameworks serve the needs of architecture-centric software development from complementing perspectives, it is a rather natural idea to combine them into an object-oriented application development environment. This idea is made concrete in a research project jointly carried out by the universities of Tampere and Helsinki, developing a pattern-based framework engineering tool called *Fred* (Framework Editor) [7]. The typical work flow when using Fred is illustrated in Figure 3. A framework designer instantiates a collection of design patterns ("contracts" in Fred terminology) over low-level elements such as components, classes and idioms into an application framework, which an application designer then completes into an application. Fred supports these activities by providing a library of documented design patterns, by

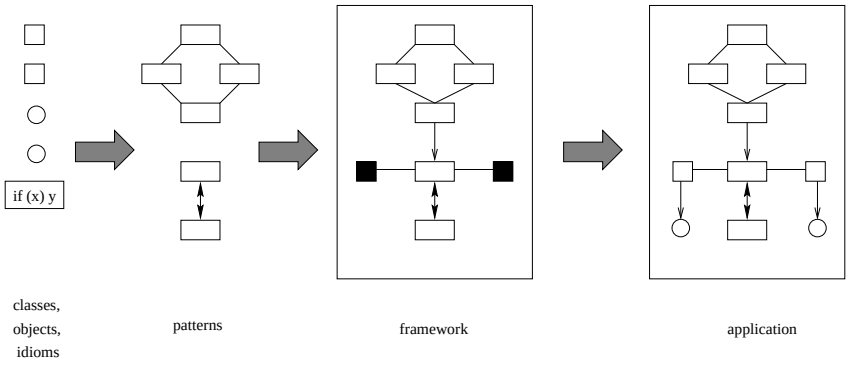

Figure 3. *From classes to applications via framework*

generating program code (in Java) from them into the framework, by keeping track of the hot spots in the framework, by guiding the application designer through the hot spots, and by giving her instructions for filling the hot spots with executable (Java) code. It is also possible to enter this work flow in the middle, e.g., by importing an application into Fred and developing it into a Fred-managed framework.

7 Reverse Architecting

Figure 3 illustrates a conventional *forward engineering* process where the flow is from design (patterns, framework) into implementation (application). When moving to the other direction, from implementation (application) to design (patterns, framework), the process in general is called *reverse engineering*. Reverse engineering is typically assisted by automatic analysis tools, but human expertise is always needed, at least when interpreting and validating the results produced by the tools.

Reverse engineering has its role in areas where the code of a software system has to be analyzed and represented at a higher level of abstraction. The most common targets of reverse engineering activities are so-called "legacy systems" that are in a constant maintenance cycle and whose only reliable documentation is the source code. Typical examples of informa-

tion that is extracted by reverse engineering are the modular structure of the software and its control and data flow.

To understand the structure and behavior of the system before modifying it, some form of reverse engineering must be applied. If the modifications are radical to the extent that the whole architecture or even the implementation language is changed, the term *re-engineering* is used. Notice, however, that reverse-engineering techniques are not categorically tied to maintenance or re-engineering phases but can also be applied during the constructing forward-engineering phase, e.g. to analyze the quality of testing by computing the test suite's code coverage.

The term *reverse architecting* refers to the special variant of reverse engineering where the main purpose is to extract the system's architecture from its source code. The result of reverse architecting can take several different forms. When reverse-engineering an object-oriented system, a natural solution is to (automatically) abstract it into UML diagrams such as class and deployment diagrams (structure), or statechart and interaction diagrams (behavior).

When reversing the flow of work in Figure 3 into a reverse-engineering process, design patterns, anti-patterns, objects, components, or idioms can be extracted from the application code. Also, the aforementioned process of generalizing an application into a framework can be seen as a form of reverse engineering. Tasks like these have applications, most notably, in quality assurance (are there any "good" or "bad" patterns in the code?), software reuse (are there any reusable entities in the code?), and object-oriented re-engineering (how could the code be transformed from conventional into equivalent object-oriented form?).

8 Product Families

A *system family* (or, a *product family*) is defined as a group of systems sharing a common, managed set of features that satisfy the core needs of a scoped domain. The motive behind a system-family approach is the potentially large-scale and rapid reuse: a new system or product can be systematically built from a common set of (software) assets. These assets can vary from low-level code-specific ones (components, functions) into high-level architectural ones (domain model, reference architecture).

The highest degree of reuse is obtained if the complete (reference)

architecture of the family is reused when developing a new product. A product-family approach does involve not only technical software assets but also a specific architectural development process that takes into account the special requirements of the (possibly unknown) future products of the family, as well as a standardized documentation notation and policy. From a high-level perspective, a family-based product development process has four main phases: (1) analysis and modeling of the domain into general requirements, (2) development of a general reference architecture for the family, (3) refinement of the general reference architecture into a product-specific architecture, and (4) development of a new product based on its architecture (and on other available reusable assets).

The idea of product families was originally introduced by Parnas already in the 1970's [11], but the approach did not reach industrial maturity until in the last few years. The main reasons for the recent success are the rise of software technologies that make architecture-centric reuse possible, the emergence of products that are quite similar in their basic functionality (e.g., mobile phones), and the constantly increasing industrial market competition. The techniques underlying a product family are typically object-oriented. For instance, the central idea behind application frameworks matches exactly the architectural needs of an evolving and extensible product family.

9 Future Trends

Software architectures are still an emerging, immature discipline. We have presented some of the areas that are under active research at the moment. There are lots of other unsolved issues that have been left out from this paper: what are the (formal) semantics of an architecture description, what is a "good" architecture, how can the quality of an architecture be measured or assessed, what is a systematic (perhaps even automated) process from requirements to architecture and further to program code, etc. The current wide interest in software architectures and in related research issues is explicitly shown by the rapidly rising number of conferences, publications, and projects in the area.

The Department of Computer Science at the University of Helsinki is actively studying research issues related to software architectures. In addition to the Fred project mentioned in Section 6, we are currently running

the projects Maisa (Metrics for Analysis and Improvement of Software Architectures) and Saara (Software Architecture Analysis, Recovery and Assessment). Maisa develops a method and a tool for the analysis of a software architecture and for the prediction of the quality of the software system developed from it, while Saara studies the problem of software equality based on architectural information. These projects are an external part of software architecture research and development at Nokia Research Center, the main industrial partner of our research group.

References

[1] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] F. Brooks, *The Mythical Man-Month—Essays on Software Engineering*. Addison-Wesley, 1975.

[3] W. J. Brown, R. C. Malveau, H. W. McCormick III, T. J. Mowbray, *AntiPatterns—Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

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

[5] J. O. Coplien, D. C. Schmidt (eds.), *Pattern Languages of Program Design*. Addison-Wesley, 1994.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] M. Hakala, J. Hautamäki, J. Tuomi, A. Viljamaa, J. Viljamaa, K. Koskimies, J. Paakki, *Task-Driven Framework Specialization*. In Proceedings of Fenno-Ugric Symposium on Software Technology (FUSST'99) (J. Penjam, ed.), Sagadi, Estonia, 1999. Technical Report 104/99, Institute of Cybernetics, Tallinn Technical University, 1999, pp. 65–74.

[8] R. Johnson, *Documenting Frameworks Using Patterns.* In Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92), Vancouver, British Columbia, Canada. ACM Press, 1992, pp. 63–76.

[9] P. B. Kruchten, *The 4 + 1 View Model of Architecture.* IEEE Software, 12(6):42–50, 1995.

[10] T. Lewis, A. Glenn, *Object-Oriented Application Frameworks.* Manning, 1995.

[11] D. L. Parnas, *On the Design and Development of Program Families.* IEEE Transactions on Software Engineering, 2(1):1–9, 1976.

[12] W. Pree, *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.

[13] E. Rechtin, *Systems Architecting—Creating & Building Complex Systems.* Prentice-Hall, 1991.

[14] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[15] M. Shaw, D. Garlan, *Software Architecture—Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.