Software architecture reconstruction

Krikhaar, R.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 1

Software Architecture

In advance of discussing software architecture reconstruction, which is the main topic of this thesis, we briefly present, in this chapter, some issues related to software architecture (amongst others definitions of architecture and architectural view models).

1.1 Introduction

In this chapter we give an overview of definitions of software architecture found in the literature. But we also discuss the importance of having a good software architecture in a software intensive system.

A software architecture must satisfy requirements from a business point of view. These business goals lead to certain objectives for software architecture, to be discussed in Section 1.4. A number of good architectural patterns which may be useful in various software systems will be presented in Section 1.5. Business goals, architectural objectives and architectural patterns are related to each other. In Section 1.6 we derive from specific business goals the related architectural objectives which, in turn, lead to certain architectural patterns. A general view on this model is illustrated in Figure 1.1.
1.2 Definitions of Software Architecture

In recent years, many definitions of software architecture have appeared in the literature. The need for modular structuring and explicit handling of product families was first discussed in the late sixties [Dij68, Par76, PCW85]. Since then, software in systems has grown tremendously in size and complexity. Although these “old” structuring principles still hold, they have to be transformed into principles for the products of today’s sizes. The term software architecture was introduced in the nineties to address, amongst other things, the up-scaling of these structuring principles.

In 1992, Perry and Wolf [PW92] gave a definition of software architecture: a set of architectural elements that have a particular form. The elements may be processing elements, data elements or connecting elements.

According to Shaw and Garlan [SG96], the architecture of a software system defines that system in terms of computational components and interactions between those components. Examples of components include clients, servers, filters and layers of a hierarchical system. Interactions between components may consist of procedure calls, shared variables, asynchronous events or piped streams.

Jacobson, Griss and Johnsons [JGJ97] stated that a software architecture describes the static organization of software in subsystems interconnected through interfaces and defines at a significant level how nodes executing those software subsystems interact with each other.

Bass et al. [BCK98] gave an “often-heard” definition: architecture is com-
Components, connectors, and constraints. Connectors are a mechanism for transferring control and data around the system. Constraints are definitions of the behaviour of components.

Many different structures are involved in an architecture of a software system. In order to organize them, models have been defined that give a certain view on software architecture. We found that the models developed by Kruchten [Kru95] and Soni et al. [SNH95] are useful in industry. In the next sections these so-called view models will be discussed, including a model that combines both view models.

**1.2.1 The 4 + 1 View Model**

Kruchten distinguishes five different views in his *4 + 1 View Model of architecture* [Kru95]. Each view addresses a specific set of concerns which are of interest for different stakeholders. Figure 1.2 (taken from [Kru95]) shows the views, the stakeholders and their concerns.
The logical view supports the functional requirements: the services a system should provide to its end users. The designers decompose the system into a set of key abstractions of the domain, which results in a domain model. Kruchten suggests to use an object-oriented style to define the logical view.

The process view addresses non-functional requirements, such as performance and availability of resources. It takes into account concurrency and distribution, system integrity and fault tolerance. In this view, the control of execution is described at several levels of abstraction.

The development view focuses on the organization of the actual software modules in a software development environment (SDE). It concerns the internal requirements related to ease of development and software management. The development view is represented by module and subsystem diagrams that show the system’s export and import relations.

The physical view also takes into account the system’s non-functional requirements. It maps the various elements identified in the logical, process and development view onto the various hardware elements. This mapping should be highly flexible and should have a minimal impact on the source code itself.

The scenarios help to demonstrate that the elements of the four views work together seamlessly. The scenarios are in some respect an abstraction of the most important requirements. A scenario acts as a driver to help designers discover architectural elements, and also helps to illustrate and validate the architecture design.

An example of a scenario is the description of the activation of follow me in a telephony switching system (activation of forward direction of incoming calls to another specified extension). Given this scenario, one can discuss how the involved processes communicate with each other via message communication, which components of the system are running, and which hardware devices are involved. The scenario view is in fact redundant with the other views, hence the “+ 1”.

The various views are not completely independent of each other. The characterization of the logical elements helps to define the proper process elements. For example, each logical element (object) is either active or passive (autonomy of objects); elements are transient or permanent (persistency of objects). The autonomy and persistency of objects have to do with the process view. The logical view and development view are very close, but address different concerns. The logical elements do not necessarily map
Figure 1.3: Relationships between the Software Architectures

one-to-one on development elements. Similar arguments hold for the relation between process view and physical view.

1.2.2 The SNH Model

Soni et al. [SNH95] have investigated a number of large systems (telecommunication, control systems, image and signal processing systems) to determine the pragmatic and concrete issues related to the role of software architecture. The structures they found in the investigated systems can be divided in several broad categories. Soni et al. distinguished five different views on architecture:

- **conceptual architecture**: describing the system in terms of its major design elements and relationships between them. Typical elements are components and connectors.
- **module (interconnection) architecture**: functional decomposition and layers, which are orthogonal structures. Typical terms are subsystems, modules, layers, imports and exports.
- **execution architecture**: describing the system’s dynamic structure. Typical elements are tasks, threads, RPC and events.
- **code architecture**: describing how the source code, binaries and libraries are organised in the development environment. Code resides in files, directories and libraries.
• *hardware architecture*: describing the hardware components and their relations as far as they are relevant for making software design decisions. Processors, memory, networks and disks are typical hardware elements.

The architectural views have relations with each other as depicted in Figure 1.3 (taken from [SNH95]). A conceptual element is *implemented* by one or more elements of the module architecture. Module elements are *assigned to* run-time elements in the execution architecture. In addition, each execution element is *implemented* by some module elements. Module elements are *implemented by* elements of the code architecture. There is also a relationship between run-time elements and executables, resource files (e.g., help texts in different natural languages) and configuration files in the code architecture.

### 1.2.3 The AV Model

Kruchten described a number of design principles for constructing elements of the various views. The SNH model was defined after an analysis of existing systems, which comprised looking at an architecture from a different angle. Nevertheless, the 4 + 1 View model and the SNH model are pretty similar. A logical decision (also suggested by [BMR96]) is to combine the good parts of the two into a new view model (see Figure 1.4). We have taken the 4 + 1 View model as a basis and integrated it with good parts of the SNH model. The new model has been baptized the *Architectural View model*, abbreviated as the AV model.

The *logical view* and *conceptual architecture* are more or less similar. In both cases, the end user is the main stakeholder. The *execution architecture* and *process view* differ only in details. Soni et al. addressed the *hardware architecture* concisely, but Kruchten stressed the *physical view* more explicitly.

The *module architecture* and *code architecture* maps on Kruchten’s *development view* of Kruchten. In our new model, we divided Kruchten’s development view into two parts: *module view* and *code view*. The stakeholders of the *module view* are the programmers. The main stakeholders of the *code view* are people who are responsible for tool support. In the new model, the source code is considered part of the *code view*.

*Scenarios* in the AV model support forward engineering as well as reverse engineering of software architectures: scenarios play a role in defining ar-
End users
• functionality

Programmers
• software management

Tool support
• configuration management

 Logical View

Module View

Code View

Scenarios

Execution View

Physical View

System integrators
• performance
• scalability
• throughput

System engineers
• system topology
• delivery
• installation

Figure 1.4: Architectural View Model
chitectural elements [Kru95], and they support the analysis of software architecture [KABC96].

The precise contents of all these views have not been described explicitly. In practice, one has to experience which elements are most important. In this thesis we focus on the module view, but the code view is also required in a supporting role. It is our intention to make the contents of the module view and code view more explicit and tangible.

1.3 Business Goals

From a business perspective the following goals can be defined for products, having impact on the software architecture within such a product [KW95, JGJ97]:

- short time-to-market;
- low cost of product;
- high productivity of organisation;
- adequate predictability of process;
- high reliability of product;
- high quality of product.

Which goals must be emphasized depends very much on the type of product. The quality of a product is very important especially for medical systems, e.g. a patient must not be exposed to too much X-ray radiation. Also important is the quality of consumer products. It is for example impossible to provide every one of the millions of television users an update of the software in their television every six months\(^1\). In the currently booming market of digital videocommunication systems it is more accepted to deliver several software updates after the first release. In this business, time-to-market has high priority as the aim is to remain ahead of one's competitors. A software architect must be aware of such trade-offs in making proper architectural decisions.

1.4 Architectural Objectives

There are many architectural objectives that justify certain architectural decisions. Bass et al. [BCK98] distinguished different, called quality at-\(^1\)Although downloading of new software to a television set is foreseen in the near future.
tributes. These quality attributes are discernable at run-time (performance, security, availability, functionality and usability) or they are not discernable at run-time (modifiability, portability, reusability, integrability and testability).

In this thesis we want to discuss architectural objectives in more abstract terms. We distinguish the following architectural objectives, which are not necessarily orthogonal:

- comprehension;
- reuse;
- evolution;
- product family.

1.4.1 Comprehension

Software changes many times during its lifetime. A developer must understand the software well to be able to modify, extend or fix a bug in the system. Approximately half of the time spent on maintenance activities concerns comprehension [PZ93]. Improvement of comprehension therefore increases a developer’s productivity.

In many cases software changes are made by developers who did not originally create the part of the software concerned. This is due to the typical lifespans of our systems, which may be decades. An original developer may in the mean time have moved on to another position or may even have left the organisation. Moreover, in view of a system’s size and complexity, several developers must often have access to the same part of the software. The nature of today’s systems makes it impossible to divide a system from the start into disjunct parts of the software that can be assigned to a single person. Comprehension of software written by other people is therefore necessary.

1.4.2 Reuse

Reuse consists of the further use or repeated use of a software artifact. Typically, reuse means that software artifacts are designed for use outside their original contexts to create new systems [JGJ97]. Proper application of reuse requires a number of precautions. Design for reuse must be explicitly addressed in an organisation to be able to reuse software. Component reuse is currently a hot topic in research and practice. In general, the number
Reusable components greatly influence productivity and quality. Reuse of software is often hard to achieve (particularly due to the *not-invented-here syndrome*); it requires a lot of investment and it must be managed explicitly to be successful. The benefits of a reuse-oriented organisation start at best, two years after introduction [JGJ97]. In a business context return-on-investment times of two years are long, especially compared with the length of time between two releases.

Reusable components can only be developed with a specified architecture in mind. For a functional equivalent component one may request different implementations depending on the architecture and/or satisfying different non-functional requirements. In the world of IC design it has long been accepted that there are different implementations for a component. In the world of software this is less accepted. For example, a component in a pipe-line architecture must behave differently from a component in an event-driven system. In a pipe-line architecture a component continuously reacts on new input data while in an event-driven system a component is triggered before it processes data. It is impossible to combine any arbitrary set of components into a new system. Garlan stressed this point as the *architectural mismatch* [GAO95].

### 1.4.3 Evolution

From a business perspective, software has come to be the most profitable part of software-intensive systems. Product features of existing systems are often related purely to software extensions. In the past, product requirements were often assumed to be stable. Today they are more dynamic and evolutionary. Requirements rapidly change and product developers must allow for this fact.

The evolution of hardware also has an impact on software. Take for example software that controls image-processing units in a medical system. One must be able to smoothly integrate a new hardware image-processing unit into a new system release or one may even replace such a unit by software. So the thought of possibly having new image-processing units in the future causes this to be explicitly covered in the software architecture. Good intuition of possible market trends helps to define software architectures that are future proof.
1.4.4 Product Family

Product family architectures are architectures especially designed to manage (and enhance) many product variations needed for different markets. For example, in different parts of the world there are different television broadcast standards, which affects the software embedded in a television. A television’s user interface is also language-dependent. Furthermore, products may also vary in the number of features they include. A television may be packed with or without an Electronic Programming Guide (EPG). One must be able to switch the EPG feature on or off in a late stage of the production process. Software architecture must be capable of facilitating all such variations, i.e. it must be flexible.

1.5 Architectural Patterns

Alexander et al. [AIS77] defined a pattern for buildings and towns as follows:

“A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

These patterns are described in a consistent and uniform style.

The notion of patterns can also be applied in the construction of software. Buschmann et al. [BMR+96] and Gamma et al. [GHJV95] used schemes to describe design patterns. Buschmann et al. categorized the patterns into the following groups:

- Architectural Pattern
- Design Pattern
- Idiom (Code Pattern)

Conceptual integrity means that the same concept is always explicitly applied for similar problems. Conceptual integrity supports developers to better understand a system and it leads to programmer independence [Bro82]. In the case of larger systems conceptual integrity is even more important. The size of the development group is larger, which means that developers spend more time communicating with each other. The application of general concepts simplifies internal communication. An architect’s task is to document these concepts, but he or she is also responsible for communicating these concepts to the development group.
A concept may also be a typical solution to a certain problem. Concepts must be defined for typical problems in each stage in the development process. Design patterns are examples of typical solutions to design problems.

To illustrate the notion of patterns, we will informally discuss three architectural patterns which are related to the module view: layering (Section 1.5.1), generic and specific components (Section 1.5.2) and aspects (Section 1.5.3). While analysing the Tele system we experienced the benefit of applying these patterns. In Chapter 6 we will return to these patterns to discuss architecture verification.

1.5.1 Layering

A layer is a group of software elements. Layers are strictly ordered. Given the ordering, higher layers may use only lower layers. We distinguish two types of layering:

- **opaque layering**: a layer is restricted to use only the layer directly below it. The idea behind opaque layering is that each layer makes an abstraction of all the layers below it and adds some extra functionality. An example of this principle is the 7 layer OSI stack\(^2\) [Tan76], illustrated in Figure 1.5.

- **transparent layering**: each layer is allowed to access services of all the layers below it see Figure 1.6. A layer abstracts functionalities in lower layers where appropriate, but it does not encapsulate functionality that has already reached a proper level of abstraction in a lower layer.

An advantage of opaque layering is that the user of a layer needs to know only the layer below it. It does not have to have any knowledge of the lowest layers. A disadvantage is that each layer must also provide functionality from the lower layer when required by a higher layer. This often leads to renaming of functions without adding any functionality. Another disadvantage is that the lower layer must have knowledge of higher layers to be able to provide proper functionality (to avoid the risk of all the non-exported functionality of the lower layers being provided again).

A transparent layer provides functionality to the outside world, without paying too much attention to the layers that use the functionality. A disadvantage is that when the interface of a layer changes, it may affect all

---

\(^2\)In a new edition of his book Tanenbaum defined a hybrid reference model with only five layers.
the higher layers.

Layering generally makes it possible to test a system incrementally. Layers can be tested one by one, starting at the bottom, i.e., when a layer of level $n$ passes the test, one can test layer $n + 1$, assuming that layers $1 \ldots n$ are functionally correct. Layers also facilitate the control of the development process and product releases.

Layers can be defined at different levels of abstraction. The following example of layering at the highest level of abstraction, i.e., subsystems, is taken from telecommunication industry [KW94]. It is very common to distinguish in a communication system the following layers, which we call subsystems (from top to bottom):

- **Service Management**: dealing with actual services of the system. In a switching system e.g., it deals with redirecting a telephone call when the *follow me* to another number feature is active.
- **Logical Resource Management**: providing logical resources. These resources are based on resources provided by *Equipment Maintenance*, but they are made hardware-independent. At this level an operator configures a communication system.
- **Equipment Maintenance**: dealing with the maintenance of peripheral hardware. It provides virtually error-free peripheral hardware to the higher subsystems (in a telecommunication system the functions of
failing hardware must be taken over by other hardware components). Hardware specifics are hidden. This subsystem provides an abstract representation of physical resources and their usability.

- *Operating System*; containing functionality provided by a normal operating system. It also provides some general functionality to higher subsystems, including e.g., software downloading, recovery and man-machine interface procedures.

### 1.5.2 Generic and Specific Components

Software components are currently a hot topic in software architecture research and practice. Szyperski used the following definition of software component [Szy97]:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only.

A software component can be deployed independently and is subject to composition by third parties.”

A component-based system consists of a number of components. One can divide these components into two kinds: *generic* and *specific*. Generic functionality, which resides in *generic components*, exists in almost all the products in the family, and specific functionality, residing in *specific components*, does not exist in all products.

Generic components represent the common part of all the products of a family. A crucial task of an architect is to distinguish generic and specific functionalities. It is not just a matter of factoring out the common functionality, because (yet unknown) future enhancements must also be taken into account.
Generic components may already be bound at compile and link time without the flexibility of configuration being adversely affected. The set of generic components form the skeleton of all products. Specific components can rely on the availability of this skeleton, but they are not allowed to rely on the availability of specific components.

This also means that only generic components can be responsible for facilitating communication between specific components (see Figure 1.7). During the system’s initialisation time, specific components announce themselves to the generics. Via a call-back mechanism the generic component is able to access the specific component at run time. A specific component can call a generic component’s functionality, on its turn, this generic component can call (via a call-back function) another specific component’s functionality.

Different types of generic components can be distinguished. A refinement of generic component types has been discussed by Wijnstra [Wij96].

Example

A public telephone switching system communicates with several other switches using different protocols and different types of lines. Each customer asks for his or her own set of hardware units and his or her own set of protocols. The system must be configured according to the user’s needs. In a late stage of the development trajectory one must still be able to con-
figure a system. It must even be possible to extend such systems (when they are running in the field) with new hardware and/or protocols. Explicit handling of generic and specific functionalities (combined with late binding) satisfies this list of requirements [KW94].

1.5.3 Aspects

In addition to object-oriented system modelling [Boo91], one can also simultaneously address a functional view on the system. In the case of large systems it is even necessary to apply another structuring mechanism for comprehension reasons. We call the means used for this structuring approach *aspects*. Before developing the separate components, one must define aspects which are in principle applicable to each component. Such a set of aspects is fixed for the whole system.

As an example we give the aspects of a typical telecommunication system:

- *normal operation*;
- *man-machine interface*;
- *recovery*;
- *configuration management*;
- *fault handling*;
- *performance observation*;
- *test*.

The notion of aspects is relevant in the various development phases. During system testing the aspects can be used to structure the process and decide on the (functional) completeness of the test. Aspects should explicitly appear in all the software artifacts (design documents, source code). A simple, but effective, implementation of aspects at source code level involves the use of prefixes (according to the aspect name) for functions, variables and files. Aspects must also be handled explicitly in design documents. For example, a reader who is interested in a certain aspect should be guided through the document in a natural fashion. This can e.g. be achieved by prescribing obligatory (sub)sections.

The *System Infrastructure Generics* (SIGs) are special generic components, which usually reside in the lowest subsystem. They deliver some basic functionality of the system. One must define (one or more) system infrastructure generics to implement the basic functionality of an aspect. For example, the man-machine interface uses basic functionality (windows, menus, etc.) which reside in SIGs. Another example is exception handling,
the basic infrastructure for achieving exception handling (e.g. popping as many return addresses from the call stack as required), is implemented in an exception handling SIG.

1.6 Relating Goals–Objectives–Patterns

In the previous sections we have discussed business goals, architectural objectives, and architectural patterns. Although the business goals are very general and hold for (almost) any business, it is obvious that some priority ordering is necessary per system (or market). Given the ordering of business goals, we can derive an ordering of architectural objectives, as illustrated in Figure 1.8. For example, the cost of product is related to the amount of reuse that can be established. Furthermore, architectural objectives can be mapped on architectural patterns. For example, when a product family is concerned it is good to explicitly distinguish generic and specific components.

Making an explicit Goals-Objectives-Patterns (GOP) diagram for your system helps to make proper trade-offs during the creation of software architectures. The GOP diagram of Figure 1.8 (simplified version of GOP diagram in [KW95]) should therefore be seen as just an example; extra goals, objectives and patterns and lines could be required for your system. The absence of a line does not necessarily mean that there is not a relationship, but it can be seen as a relative unimportant relation.

1.7 Final Remarks

Most of the discussed issues stem from the Building Block Method used in Nuremberg for the development of telephony switching systems (Tele). The Building Block Method and its application to large systems have been discussed in a number of reports [KW94, KL94, Kri94, Kri95, LM95, Wij96].

We have addressed only a few architectural patterns of the module view. Other good architectural patterns for this view exist, but the other views on architecture should also be covered with architectural patterns. In this chapter it has been our intention to give a non-exhaustive overview of the variety of issues relating to software architecture.
Figure 1.8: Goals, Objectives and Patterns