Towards a Framework for Large Scale Quality Architecture

Markus Voss, Andreas Hess, Bernhard Humm

Abstract—sd&m Research is actively working on “Quasar Enterprise” – a framework of architectural principles and references for large scale quality architecture. In this contribution we present its basic and advanced concepts plus a concrete set of rules for architectural design on enterprise level. By explicitly working out these concepts and rules we can get a lot more tangible on semantics of the hyped concept of service-oriented architecture (SOA).

Index Terms—Application Landscapes, Quality Architecture, Service-Oriented Architecture (SOA)

I. INTRODUCTION

Over the last years, sd&m Research\(^1\) has defined “Quasar” (Quality Software Architecture) – a well founded framework of terms, architectural principles and references for careful planning and robust construction of information systems \([1,2,3]\). Quasar has evolved to be a big success story since its principles and references have been utilized in dozens of industrial bespoke application development projects at sd&m and also influenced thinking in the German software engineering community quite a bit.

On the other hand, pure bespoke application development projects tend to decrease in relevance compared to projects where component integration and enterprise level architectural design are of prime importance. An architectural concept much discussed in this context recently is that of Service-Oriented Architecture (SOA). SOA promises to be a good way to structure systems on a large scale in order to achieve agility, flexibility and maintainability – properties of IT solutions needed to address ever faster changing requirements of business and cost pressure. Much has been written about SOA \([4,5,6,7,8]\) and sd&m has published some of its experiences with SOA in industrial practice, too \([9,10,11]\). Studying these contributions, one finds that on a very general level the understanding of SOA is quite unitary. The concept of loose coupling e.g. is considered central in all contributions. Beyond that however, one can make two observations:

• First, the in depth understanding of SOA is quite different, which already starts with the usage of terms.

  • Secondly, if it comes to precise rules for designing components and services we haven’t found much in the given literature that goes beyond the general concepts.

Therefore sd&m Research over a year ago has started work on “Quasar Enterprise”. As with “Quasar” sd&m Research aims at a well founded framework of terms, architectural principles and references for careful planning and robust construction – but this time for architectural work on a large scale. “Quasar Enterprise” – when completely developed – is meant to be our enterprise architects’ handbook like \([3]\) is today for our software engineers.

Putting together “Quasar Enterprise” it is important, not to “reinvent the wheel” and search for a completely new theory of large scale architecture. The idea is to dig out proven best practices from our enterprise level architecture and systems integration projects and to match this with the state of the art in scientific discourse to come up with a real synthesis. Today \([12]\) is the most comprehensive coverage of “Quasar Enterprise” in this respect. But even though there is still some way to go to finish “Quasar Enterprise”, we think it is worth to share our view of some of its central elements with the community.

In the following we structure and introduce these in three categories:

• The Terms and Basic Concepts: These are the basics to build the framework on. The contribution of “Quasar Enterprise” here is simply to standardize by clear definition. Within this paper we present some of the central concepts of the ontology.

• The Advanced Concepts: These are the architectural ideas and tools to be used to further specify Quality Architecture compared to what is possible with only the general concepts in terms of the least common denominator of what a SOA is. Here the contribution of “Quasar Enterprise” is to dig these out and make them

\(^1\)sd&m Research is the research and technology management unit of the sd&m AG – a German software development, systems integration and IT consulting company of over 1.000 IT professionals. sd&m’s focus is on developing and integrating custom solutions.
explicit. Some of these will be outlined in this paper.

- The Rules: These are the actual guidelines on how to design domains, components, services and couplings. Here the contribution of “Quasar Enterprise” will be to put these together in a comprehensive depiction – possibly for the first time. In this paper we present the current draft of the core set of rules and even though we still work on further consolidation, we regard it valuable to present this intermediary result as basis for more in depth discussion with the architectural community.

II. TERMS AND BASIC CONCEPTS

A. Business Architecture and Application Architecture

Different from many approaches to SOA which are technology-driven (e.g. the approaches of many tool vendors, who regard concrete enterprise service bus (ESB) solutions as core), we regard SOA is a business-driven approach to quality architecture. The success factors for reaching the goals of agility and flexibility typically lie within aligning the architecture of the application landscape according to characteristics of the business domains and the business processes. Therefore it is obvious, that a clear distinction between the terms and concepts of business architecture and application architecture is crucial for a sound approach. Doing so, we actually follow the structure that most of the so called Enterprise Architecture Frameworks impose (e.g. Zachman [13], TOGAF [14] or Cappemini’s IAF [15]). In [12] we go into more detail on the relationship between these frameworks and “Quasar Enterprise”.

For the terms of business architecture we define as follows:

- We define business architecture to be the set of all propositions and regularities about business domains, business processes, business services and business objects as well as about their relationships amongst each other.
- A business domain is a segment of an enterprise that has influence on processes and organization, e.g. market, customers, suppliers, product or service type, subsidiary structure or sales channel. To shape an enterprise’s IT landscape it is important to know, which of the domains are essential and differentiating, since these lead to identify appropriate application domains (see below).
- A business process is a sequence of tasks to reach a defined business goal directly or indirectly connected with the enterprise’s products or services. It may be organized in sub-processes, the smallest units of which are elementary tasks that are to be carried out non-interruptible, by one actor, in one place.

- A business service is such an elementary task, which is context free, has a unique actor and a well-defined business goal connected to its execution. It is these business services, that a SOA should be built on in terms of its application services (see below) being deduced from these business services.
- Business objects finally are real world objects– material or immaterial. For IT architectural problems we usually work with their model representations which we call information objects.

On the other hand the actual subject of our work as enterprise architects or systems integrators is a company’s application architecture. For the terms here we define as follows:

- Application architecture is the set of all propositions and regularities about application domains, application components, application services and application service-operations as well as about their relationships amongst each other.
- An application domain is a set of application components belonging together to be regarded as a unit in conjunction with shaping the application landscape.
- The application landscape is the entirety of all application components of an enterprise together with their interconnections in terms of interfaces and data.
- According to [3], a component is an essential element of design, implementation, and planning. It exports and imports interfaces, hides its implementation details, and can be hierarchically structured. There are many other definitions around of what a component is (e.g. [16]), and the concept is general enough to cover the complete spectrum from very large scale components (complete application landscapes) to very small scale components (single software objects or classes). In the context of enterprise level architecture we use the term application component to denote a self-contained unit of functionality.
and data, belonging to an application domain and exporting specific parts of its functionality as an application service.

- An application is any collection of application components regarded meaningful by system designers.
- An application service corresponds with the interface of an application component. It allows access to a defined functionality which can be utilized as part of a comprehensive process. The application service therewith yields an abstraction of the exporting application component hiding its implementation details. It is self-contained and context-independent. The definition of an application service has the character of a contractual agreement between service provider and service user covering syntax, semantics and non functional properties.
- Finally, application service-operations are the actual functionalities application services are made of. They have a signature in terms of input and output types and semantics specifiable by pre- and post-conditions.

Fig. 2 shows the most important terms introduced and their relationships.

In the context of enterprise level architecture and when misapprehension is avoided, we only speak of domain, component, service and operation. Fig. 3 shows these elements of application architecture.

B. Service-Oriented Architecture and Quality Architecture

Having the terms defined, one can now formally say, what a Service-Oriented Architecture is. The least common denominator of all the definitions around probably is the following set of statements (defined e.g. in [17]):

- Functionality is encapsulated in services
- Services represent publicly known interfaces
- Services are loosely coupled
- Services are atomic

It is obvious, that these attributes of the architecture are not sufficient to guarantee quality in the sense of the architecture achieving a high ranking in typical quality measures like understandability, adaptability, flexibility, agility, maintainability, manageability, sustainability, cost efficiency, security, etc. Quality Architecture is achieved by constructively searching for an optimum among these measures according to the actual stakeholders’ needs. A framework for Quality Architecture like the one we are after with “Quasar Enterprise” therefore needs to assist in this process with more advanced concepts and tools as well as with concrete rules.

III. ADVANCED CONCEPTS

A. Lay-out Plan

The lay-out plan is an architectural tool to analyze and shape application landscapes. It is used to both visualize the as-is structure of the landscape and visualize, how it is planned to be developed in the future. By highlighting two dimensions of the business architecture – often the prime business processes and some other important business domain - it spans an area in which the elements of the application architecture - application domains and components - can be positioned according to how they address or will address the according segments of these domains. Fig. 4 shows an example of a lay-out plan.

Lay-out plans are nothing new and widely used, e.g. as a tool in so called business-IT alignment projects. Exactly this is why they are useful in Quality Architecture. They bridge between business and application architecture and support a business-driven approach. In 4.1 we show, how rules with respect to the development plan can help find domains.

B. Service Categories

A second valuable architectural tool is the concept of service categorization. Application service-operations are categorized according to their nature in terms of which segment of large scale architecture they primarily support. Related concepts can be found in the literature, e.g. in [8]. From our expertise we found, that the most promising set of categories in terms of quality architecture is the following:
Interaction: Service-operations for user interaction with the application landscape (e.g. in enterprise portals)
Process: Service-operations supplying implemented business processes (e.g. in order management)
Function: Service-operations supplying implemented core business functions (e.g. in billing)
Inventory: Service-operations for managing and accessing business data (e.g. in customer management)

The categorization of service-operations leads to an ordering concept for application services and components. For the later, the ideal of every component being of exactly one category leads to a technical reference architecture on enterprise level. This is shown in Fig. 5.

Note that the concept of categorized application components is not to be confused with the layers or tiers of a single application, often called presentation, business-logic and data-handling. Large scale application components – even though they may be of only one service category – usually consist of sub-components on all three layers. One may only state, that components of a special category are typically “heavier” in the according layers (meaning that they have “more” or “bigger” sub-components within these layers). Fig. 6 illustrates this.

C. Integration Levels

The third architectural tool of importance is the concept of integration levels. The basic idea is that application components can be integrated in three forms resembling the layers or tiers of an application as mentioned above:

- Presentation integration: Components are coupled in terms of their presentation layer components interacting. This sort of coupling is typically technically implemented by portal server or screen scraping technologies.
- Service integration: Components are coupled in terms of their business-logic layer components interacting. This sort of coupling is typically technically implemented by enterprise application integration (EAI) technologies.
- Data integration: Components are coupled in terms of their data-handling layer components interacting. This sort of coupling is typically technically implemented by data integration and extract-transform-load (ETL) technologies.

Fig. 7 illustrates the concept.

The concept of integration levels is also not new. One can find e.g. the same levels in SAPs Netweaver architecture [18] (alternatively named people integration, application integration and data integration).

As we see, integration is nothing more than the enterprise level term for coupling of application components or applications. Note that the concept of integration levels is not to be confused with the service categories since these are strictly orthogonal. As above, one may only state, that components of a special category are typically “heavier” coupled in the according layers in that they have “more” or “closer” interactions within these levels. Fig. 8 illustrates this.

Talking of Service-Oriented Architecture, the focus usually is on the service integration level. EAI / ESB solutions are often regarded as the typical means for implementing a SOA technically.
Talking of Quality Architecture however, choosing the appropriate levels of integration is indeed a measure.

D. Business Process Management and Orchestration

One specific form of service integration as introduced above is by following a business process management (BPM) approach. BPM recently is often cited in the same breath with SOA (e.g. [19,20]). Even though – as with SOA – BPM can be regarded as (only) an IT management approach, here we regard it as a specialization of the architectural style of SOA. The BPM approach aims at a SOA, where all elementary steps of a business process are supported by services and all the comprehensive process logic and the tasks of coordination between these elementary services is located in another dedicated service.

Another term of interest in this context is orchestration. According to [11] e.g. orchestration is the definition of a business process as a composition of elementary services according to the BPM approach as defined above. In the contrary case where process execution logic is not centralized but implicitly contained in the messages exchanged between the elementary services themselves the composition is called choreography.

The BPM structure allows for separating the more stable elements in terms of the elementary functions from the more agile ones dealing with potentially fast changing business processes. According to the concept of service categorization the former correspond to function or inventory components/services. The later correspond to process components/services and their implementation can be based on either dedicated systems like e.g. order managements solutions allowing for the flexible configuration of workflows or can be based on dedicated BPM products. How to actually implement a BPM-aligned SOA therefore is an explicit design decision.

IV. Rules

A. Designing Domains

Business domains are defined by business requirements. Therefore business domains in the first place are found along the core processes of the value chain. Other domains may result from core business objects like partner or product that are of importance cross the business process. Thirdly, business domains of importance may be found looking at the single branches, e.g. product type. However, for business domains there is no rule of how to find the right ones for your architectural approach other then to try to understand the business of the enterprise and its strategy. Business domains result from questioning the stake holders.

This is different with application domains. Well cut application domains follow the business domains identified as important. For optimizing the domains design the following rule applies:

- **Optimal Covering**: Design and arrange your application components within domains in a way, that according to the most important business domains the covering in the lay-out plan is optimal. The covering is optimal, if there are no avoidable multiply covered areas and if horizontal and vertical partitioning is as homogeneous as possible.

Unavoidableness and possibility in this regard result from given limitations like e.g. organizational constraints (accountability). Obviously, since shaping an application landscape on enterprise level always starts from some a given as-is status, applying the rule in reality is about planning a migration on a large scale.

B. Designing Components

Finding components is probably the oldest design challenge in software engineering. Nobody will be surprised, that the “golden rules of good design” like

- Information Hiding (as well as the related concepts of encapsulation and minimal coupling / maximal cohesion),
- Separation of Concerns
- Decoupling

also apply on the enterprise level finding the right application components. Taking these plus the advanced concepts presented in chapter 3 plus our findings from industrial practice we propose 7 concrete rules for designing application components in large scale quality architectures:

- **Professional Categorization**: Define your application components according to appropriate professional categories.

On the highest level this demand for a separation of concerns corresponds with the demand for a unique mapping of application components to domains (see 4.1). But on lower levels it also calls for separating components inside one application domain according to different stakeholders involved, different speeds of change (see 3.4), different dependency on process etc.

- **Service Category Orientation**: Define your application components to be of exactly one service category.

After what was said in Chapter 3, this rule is obvious. We really vote for taking service categories serious, since mixtures here have often proven to be time and cost intensive to maintain and later replace.

- **Dependencies according to Service Categories**: Allow only for dependencies from components to components on equal or lower level of service category.

Dependencies amongst components can be of type “knows”, “calls”, or “gets data from”. For all these we demand: components of type interaction only depend on components of type interaction, process, function or inventory, process only on process, function or inventory, function only on function or inventory and inventory only on inventory.
No Cyclic Dependencies: Avoid cyclic dependencies amongst application components. Use (if appropriate) merging, separation or call-back

Cyclic dependencies cause problems in development, testing, maintenance and replacement.

Minimal Coupling / Maximal Cohesion: Design your components for minimal coupling and maximal cohesion

This is well known from software design and rightfully demanded on the large scale, too.

Data Sovereignty: Access to all business objects data is allowed through components of type inventory only. The areas of business objects data to be handled by different components are to be disjoint

The same argument counts as with cyclic dependencies.

Manageable Size: Domains should have a medium-size number of components (not too many, not too little) and all components of a domain should have approx. equal size (e.g. in terms of lines of code, use case points etc.)

This is a well known principle supporting manageability of the architecture.

C. Designing Services

Designing the components is one side of the coin – designing the services is the other. But before we can define the rules for the later, we need to add a little enhancement to the definitions from Chapter 2 concerning types of application service-operations. Operations can be classified as follows:

- Elementary: These operations yield simple basic functionality.
- Compound: These operations are implemented by using multiple elementary operations and provide added value services.
- Orchestrateable: These operations are designed to be a basis for an orchestration as defined in chapter 3. Usually these are compound.

Therewith we define the following 7 rules for the design of service-operations:

- Service Category Orientation: Every service-operation should be of exactly one service category

  Same argument as for components.

- Implementation Neutrality: Service-operations should not export any implementation details

This is the obvious demand for encapsulation, e.g. technical keys should never be part of any operation signature.

- Normality (being complete and free of redundancy): Elementary service-operations should be normal

  For a detailed discourse see [21].

- Coarse-Granularity: Orchestrateable service-operations should be coarse-granular

  Few operations doing much are better than many operations doing little in the context of loose coupling.

- Idempotency: Orchestrateable service-operations should be idempotent

  Idempotent operations can be called multiple times if necessary not changing the result. This is necessary in the context of decoupling.

- Context-Insensitivity: Service-operations of type function and inventory should not have any knowledge about the context in which they are called and therefore should not make any assumptions about it

  This is another necessary condition for decoupling.

- Constraint on Transactionality: Transactional behavior (guaranteeing atomicity, consistency, isolation, and durability) is only allowed for operations of category inventory or function. All other operations are to be designed non-transactional with according compensation operations

  And this is yet another necessary condition for decoupling.

All presented information about rules for designing services are summarized in Fig. 9.

![Fig. 9 Rules for Designing Services](image-url)
D. Designing Couplings

Finally, the rules for designing coupling will be presented. We order these in two groups. The first group contains the general rules of coupling.

- **Loose Coupling:** Couple your application components as loose as possible and as tight as necessary

This is the way to achieve maximum flexibility.

- **Constraint on Tight Coupling:** The tightest form of coupling in terms of synchronous intra-process communication is only allowed for operations of category inventory or function

The second group finally contains rules with respect to the integration levels. These are self-explanatory.

- **Presentation Integration:** Choose presentation integration, if user interfaces to be integrated already exist and integration logic is restricted to dialogue control and value passing

- **Service Integration:** Choose service integration, if applications to be integrated with substantial business-logic already exist and quick implementation of new processes and products is strategic

- **Data Integration:** Choose data integration, if data redundancy already exists or is planned for performance reasons and differences between logical and physical data model are small

V. PROSPECT

In this contribution we have outlined some of the core elements of what we aim at with “Quasar Enterprise” – a framework for Quality Architecture on the enterprise level. It was shown that SOA is indeed a promising concept, but also that there is more to Quality Architecture than SOA. Most prominently we have proposed a draft set of concrete rules to design a SOA to meet usual quality requirements.

Future research regarding the presented elements of “Quasar Enterprise” will be on further consolidating the set of rules by extending the set of concrete sd&m projects and experiences to validate the rules against. Also we will elaborate more on the “traceability” between rules or design decisions and the actual quality attributed listed e.g. in II B.

Many other elements that “Quasar Enterprise” will be about were not mentioned at all like the idea of reference architectures for specific parts of the application architecture (many of them already existing at sd&m, see e.g. [22]) or the concept of embedding as a mechanism for integrating legacy systems and COTS application components into a SOA.

Nonetheless we hope that this contribution will lead to the sought-after scientific discourse within the community.

REFERENCES