What is Software Architecture?

Fritz Solms
Department of Computer Science
University of Pretoria
South Africa
fsolms@cs.up.ac.za

ABSTRACT

Currently there is no consensus on what exactly software architecture is or where the boundary between software architecture design and application design lies. In this paper the concept of a software architecture is analyzed from a number of different perspectives. In particular, it is argued that a software reference architecture is a relatively pure specification of a software architecture as it addresses only infrastructural and quality concerns and does not provide any application functionality. Two examples of widely adopted reference architectures are analyzed in order to gain a deeper understanding of the concept of a software architecture. Current definitions of software architecture are analyzed and three classes of software architecture definitions are identified. The differences of the concept of software architecture across these classes is discussed and their relationships to current architecture description languages are explored. It is argued that none these definitions provide a sound basis for differentiating between application and architecture components. Based on insights obtained from the analyzed reference architectures, the paper proposes a new definition of software architecture as well as a specification of the elements of a software architecture specification. Software architecture is defined as *the software infrastructure within which application components providing user functionality can be specified, deployed and executed.* It is argued that a software architecture description should include, across levels of granularity, the basic concepts and constraints within which application components are to be specified, the architectural components addressing technical concerns, the integration infrastructure, and the architectural strategies used to concretely address quality requirements.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture

General Terms

Design

Keywords

software architecture, reference architecture

1. INTRODUCTION

The responsibilities of a software architect include the design, documentation, assessment and evolution of a software architecture within which functionality can be deployed and executed in such a way that the quality requirements for the system are met [5]. It is understood that the software architecture specification contains many of the important design decisions and that these decisions are hard to change at a later stage [14].

However, even though software architecture is not a new field, we do not yet have consensus on what exactly software architecture is. We have a range of definitions which differ in some significant ways. Most of definitions are such that the boundary between architecture and application design is left open for interpretation. *Application design* is defined here as the design of the components addressing the functional requirements of the user. The resultant application components are deployed in a software architecture addressing the architectural requirements including the quality requirements for the system.

The different approaches to software architecture manifest themselves in a wide range of *Architecture Description Languages* (ADLs) which differ in what aspects of a software architecture they can specify. Furthermore, one's choice of a definition of software architecture influences the content of an architectural requirements specification, how the architectural design is done and its deliverables, and how software architectures are to be assessed. Consequently the choice of definition of software architecture directly impacts on the responsibilities of and skills requirements for software architects. There would thus be significant benefits if the software development community reached consensus on the concept of software architecture.

The remainder of the paper is organized as follows. In section 2 some general insights into software architecture are discussed. It is argued that reference architectures are relatively pure representations of software architectures and that these can be studied in order to deepen one's understanding of the concept of software architecture. Two of the more widely used reference architectures are introduced in order to analyze various aspects of software architecture and to highlight the boundary between software architecture and application functionality. In section 3 different classes of software architecture definitions are identified and analyzed. A new definition of software architecture is proposed in section 4. The corresponding elements of a software architecture specification is discussed and it is highlighted that these are present across levels of granularity. The section also includes a discussion around the interplay between architecture and application design. In section 5 the work is put into context of some related work. Finally some conclusions are drawn and outlook for future work is discussed.

2. SOME INSIGHTS INTO SOFTWARE ARCHITECTURE

In this section we study software architectures from various perspectives in order to assimilate an understanding of the concept of software architecture. Based on the insights discussed in this section, we will propose a definition for the concept of software architecture and identify its elements.

2.1 Reference architectures

Since we currently do not have consensus on the definition of software architecture, we also do not have a single definition for a reference architecture. Lloyd and Galambos define [19] reference architectures as *domain-specific architectural templates which aim to address the architectural concerns for a particular class of problems.* This is the definition chosen for the purpose of this paper.

For example, a gaming platform might provide a suitable referance architecture into which a certain class of computer games can be deployed into, whilst a reference architecture for enterprise systems addresses the infrastructure and quality concerns of a certain class of enterprise systems.

Zhu, Staples and Tosic, on the other hand view reference architectures as template systems with implemented template business processes for specific business domains[31]. This paper differentiates between a reference architectures which address the infrastructural and quality concerns of a class of system, and a reference applications which provide template application functionality.

Examples of widely used reference architectures for enterprise systems include the *Java-EE* (Java Enterprise Edition) reference architecture as well as services oriented architectures (SOAs). These reference architectures have simplified the responsibility of architecture design for typical enterprise systems by providing a template solution which addresses common architectural concerns for such systems.

A reference architecture does not contain application logic. It provides an infrastructure within which application logic can be deployed and executed. As such they provide examples of relatively pure software architecture specifications for which the separation of application and architecture logic has been enforced.

An implementation of a reference architecture is called a *framework*. A framework provides a concrete implementation of the architectural elements and strategies specified by a reference architecture. A specific system architecture based on the template provided by the reference architecture is called an *instance architecture*. Such an instance architecture may or may not be realized within a framework.

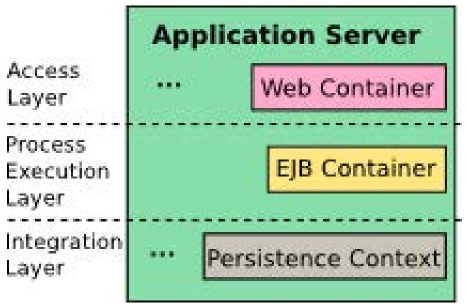

Figure 1: At the first level of granularity the infrastructure of the application server is based on the layered architectural pattern.

In this paper we use these two commonly adopted reference architectures to discuss the concept of a software architecture.

2.1.1 *Short overview of the Java-EE reference architecture*

Java-EE was introduced in 1999 via a community process led by Sun Microsystems. At the time of writing the current version was version 6. Java-EE is implemented in a wide range of open-source and proprietary frameworks including *JBoss*, *Apache Geronimo*, *IBM Websphere* and *Oracle WebLogic Application Server*. These Java-EE frameworks

- enforce a layered architectural style containing the architectural layers defined in the Java-EE reference architecture,
- and provide
  - access and integration infrastructures,
  - a persistence infrastructure for storing, retrieving and updating objects which are ultimately stored in relational databases, and
  - a set of implemented architectural strategies[1] to address quality requirements for typical enterprise systems, including reliability, scalability, security, performance, and integrability with a particular emphasis on the first three.

The main architectural element of the Java-EE reference architecture is an application server within which applications addressing the functional requirements of application users are realized. The architectural pattern[2] used at this highest level of granularity is the layered architectural pattern (see figure 2.1.1). Application servers provide an access layer, an process execution layer, and an integration layer which provides, amongst other things, integration to relational databases.

[1]Architectural strategies are also know as architectural tactics.

[2]Architectural patterns are sometimes called architectural styles.

The access layer contains a web container which hosts human and system adapters in the form of web front-ends and a web services broker. System access can also be provided through message queues, RMI (the *Java Remote Method Invocation* interface based on CORBA, the *Common Object Request Broker Architecture*) and CORBA.

The process execution layer is represented by an EJB (Enterprise Java Beans) container. It hosts the application components which provide the application functionality. In Java-EE these components are represented by *enterprise beans* which are either stateless and stateful session beans or message driven beans. The Java-EE specification requires that the components are decoupled via component interfaces and dependency injection. EJB containers maintain and optimize thread pools for these application components.

The integration layer contains adapters to other systems. One of these is the persistence context which provides the integration channel to a database. Enterprise systems commonly use relational database management systems (DBMSs) as their preferred persistence technology. The persistence context maintains an object cash and uses an object-relational mapper to map domain objects used in the processes layer onto persistent storage, i.e. a database.

Java-EE frameworks use a range of architectural strategies to address common quality requirements of typical enterprise systems. Scalability and reliability is addressed through support for clustering, thread and connection pooling, object/data caching, transactions, and messaging. Security is addressed through authentication, authorization and encryption support. Modifiability is enhanced through enforced decoupling through interfaces, dependency injection, the ability to use interception to add further functionality to existing functionality, and hot deployment (the ability to deploy, update and undeploy application functionality at run-time). Interfaces and dependency injection improve flexibility by facilitating pluggability of components across levels of granularity. Integrability is addressed by supporting standard integration technologies like CORBA and web services and providing features which automate access to application functionality through different integration channels. In Java-EE one can simply annotate a class operation as a web service. The architecture can automatically generate the web services contract, and the infrastructure for making the service available through SOAP/HTTP. Performance is improved through thread pooling, object and data caching, and connection pooling.

In Java-EE effort has been made to keep architectural concerns out of the application logic. Architectural information (e.g. whether a service should be made available as a web service) is not specified in code, but instead through meta-data which is provided either in separate deployment descriptors or in the form of annotations.

The architecture also introduces concepts and constraints for application development. At the high level, Java-EE applications are also based on the layered architectural pattern with the layers corresponding to the infrastructural layers of the application server itself. This is depicted in figure 2.1.1.

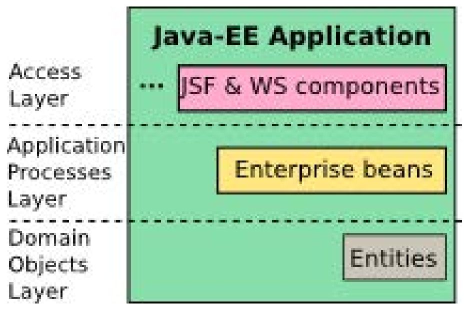

Figure 2: At the first level of granularity Java-EE application structures are constrained by the reference architecture enforcing a layered architectural pattern on the application structure.

The Java-EE reference architecture enforces, at lower levels of granularity, further application structure constraints. For example, application elements in the presentation layer are guided JSF to the Model-View-Controller (MVC) pattern.

2.1.2 Short overview of a SOA-based reference architecture

Services-Oriented Architectures (SOAs) are commonly used to provide an integration infrastructure between enterprise systems of larger organizations. Whilst there is no authoritative specification of the SOA reference architecture, there is nevertheless general agreement on the core components of a SOA. Most of the lower level elements are based on public standards maintained by the *World-Wide-Web Consortium*, W3C. The *Java Business Integration* reference architecture (JBI) is a concrete reference specification of a SOA reference architecture. *Apache Servicemix* and *Open-ESB* are two open-source frameworks which implement the JBI SOA specification.

Like Java-EE, a SOA targets the domain of enterprise systems. But whilst Java-EE puts the main emphasis on scalability, reliability and security, SOA puts the main emphasis on integrability and modifiability.

At the highest level of granularity, SOA is based on the mikrokernel architectural pattern (see figure 2.1.2. The mikrokernel itself is represented by a services bus which provides an integration infrastructure between service providers and service consumers. It provides routing of services requests to different service providers (servers) as well as an adapter layer to adapt to these service providers. Internal servers are systems within the organization whilst external servers provide the services which are sourced from outside the organization.

SOA also specifies the existence of a services registry where service contracts and services which realize these contracts are registered. Process specification languages like the *Business Process Specification Language* (BPEL) are used to "orchestrate" higher level services across lower level services. The processes are executed within process execution engines which can interpret the process specification language used. These process engines are commonly referred to as *Business Process Execution Engines* (BPEE).

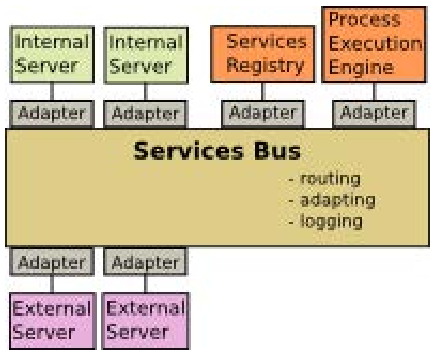

Figure 3: At the first level of granularity the infrastructure of the application server is based on the layered architectural pattern.

The BPEEs are deployed within a lower level software architecture providing the actual execution infrastructure like access to threads and memory. Common examples are EJB or Spring containers.

Like Java-EE, SOA based architectures use a range of architectural strategies to concretely address quality requirements. Recently we have also seen increasing attention on defining self-adapting SOA-based architectures which dynamically choose different architectural strategies to achieve the required quality of service [22, 4, 21].

Component based architectures require that application logic should be deployed as pluggable components which potentially maintain state across service requests. SOA, on the other hand, requires that application logic should be specified within stateless services. Application logic is specified within the *pipes-and-filters* architectural style, i.e. higher level services are assembled ("orchestrated") from lower level services by connecting them through pipes. Service may not have a direct dependency on each other. These services realize service contracts which are specified in the *Web Services Description Language*, WSDL.

2.2 Architecture functionality versus application functionality

In order to clarify the concept of software architecture, we introduce the concepts of application and architecture functionality.

We define *application functionality* as the functionality required to address the user's functional requirements, i.e. functional requirements in the problem domain and *architecture functionality* as the functionality required to address technical concerns like, for example, the functionality required to implement architectural strategies (resource pooling, load balancing, ...) addressing quality requirements or the functionality within the persistence or integration infrastructure.

Note that both reference architectures contain extensive architectural functionality, but that neither aims to address any user functionality (like the "business" logic for processing a withdrawal from an account).

2.3 Concerns of software architecture

Both reference architectures discussed in section 2.1 specify

- a set of *basic concepts and constraints* within which application functionality is to be specified and integrated.
- *architectural components* addressing technical concerns,
- *access and integration channels* through which the application functionality can be accessed and through which application logic can access externally provided functionality, and
- *architectural strategies* through which quality requirements are concretely addressed.

2.3.1 Basic concepts and constraints

A software architecture may introduce architecture-specific concepts and may specify constraints around these concepts. The concepts and constraints are often specified in the form of architectural styles or patterns with concepts which are specific to the style or pattern.

The highest level of granularity of Java-EE specifies that application logic should be specified across three layers, a presentation layer containing web application, a (business) processes layer containing the business/domain logic as enterprise beans and a domain objects layer containing entities. It is important that there are not only concepts and constraints in the form of architectural patterns for the application code, but that the architectural components themselves may be organized within architectural styles or patterns. For example, at the highest level of granularity, both the architectural and application components are constrained within hierarchical patterns. Whilst the application logic is constrained to a presentation layer, a business logic and a domain objects layer, the high-level architectural components are an access layer containing the web container, a processing layer containing the EJB container and an infrastructure layer containing the persistence manager and integration components.

On the other hand, the basic application components of an architecture based on a *Services-Oriented Architecture* reference architecture a *service contract* and a *service* realizing that service contract. Constraints include that every service must be stateless (no state is maintained across service requests), that it is self-healing (i.e. failure in the rendering of one service should not impact on the ability to realize the next service request), and that services are assembled from lower level services within a pipes and filters architectural style.

2.3.2 Architectural components

Both reference architectures specify a set of architectural components addressing technical concerns. In Java-EE the high-level architectural component is an application server. The next lower level components are the web container providing web-based access, the EJB container providing an execution environment and entity manager providing access to relational databases.

SOA defines as core architectural components a services bus, a service registry for registering service contracts and service providers, and a business process execution engine for executing business processes.

These components provide standard APIs. For example, an application server implements a standard API for deploying applications and any implementation of an entity manager implements the Java Persistence API.

2.3.3 Access and integration channels

Both reference architectures specify access and integration channels for

- accessing functionality deployed within the architecture,
- for deployed application logic to access externally defined functionality, and
- integration channels between the architectural components defined for the architecture.

For example, Java-EE specifies a web-based access channel for human users, and web-services, CORBA and RMI channels for systems. Access between application components is via local or remote Java calls.

In SOA-based architectures access between system components is generally standardized as either synchronous or asynchronous web services access. The latter is generally via message queues.

2.3.4 Implement architectural strategies

Frameworks based on either of the reference architectures, provide an implementation of a range of architectural strategies in order to address quality requirements.

For example, they commonly implement clustering, a strategy which groups computers together in such a way that they behave like a single computer. Requests are load is distributed across the different computers within the cluster which process these requests concurrently. If a computer within a cluster fails, no further requests are channeled to it. Clustering can thus be used to improve scalability, performance and reliability.

Architectural strategies are also implemented at lower levels of granularity. For example EJB containers use thread pooling, Java connectors use connection pooling and entity managers use object caching to achieve a higher level of scalability and performance.

2.4 Architecture across levels of granularity

Often software architecture is viewed as a high level abstraction of a software system. Note, however, that both reference architectures analyzed in this paper provide not only a high-level infrastructure within which application logic can be deployed and executed, but an infrastructure which spans across a large number of levels of granularity. The architectural concerns discussed in section 2.3 are present across levels of granularity. We may thus have architectural architectural concepts, constraints (e.g. styles/patterns), architectural components, integration channels and strategies at any level of granularity.

Let us look, for example, at Java-EE. At the highest level of granularity represented by the application server, Java EE introduces

- the layered architectural style with the concepts of access, processing and infrastructure layers,
- the web and EJB containers and entity manager as components of the application server,
- integration channels between these components,
- architectural strategies like clustering, and
- infrastructural functionality like that of deploying and undeploying applications.

At the next lower level of granularity we have, for example, the EJB container which introduces the concepts of enterprise beans as business logic components. At this lower level of granularity it uses the flyweight pattern to achieve scalability across a potentially large number of stateful components. There is lower level functionality like those of obtaining a session context or an entity manager. At even lower levels of granularity there are the details of the thread pooling, the implementation of the interception framework and so on.

The basic concepts and constraints for application logic are also specified across levels of granularity. We saw that Java EE requires application logic to be specified across a presentation layer, a processes layer and a domain objects layer. The presentation layer is represented by a web application. At the next lower level of granularity, Java-EE constrains the web application to be based on the MVC pattern, introducing facelets for the views, backing beans for the models maintaining the state of the views, and binding beans for the controllers which act on user events.

Note, however, that none of the above architectural components represent objects from the problem domain. Also, none the above functionality addresses any of the functional requirements from the problem domain.

2.5 Architecture is a matter of perspective

It needs to be pointed out, though, that whether something is architecture or application functionality depends on what application one is developing.

Consider, for example, the scenario depicted in figure 2.5. If one is developing, say, a banking system, then the application functionality would include those of opening and closing accounts, processing transaction, and so on. In order to address the non-functional requirements of the application, one may choose to use a Java-EE based software architecture which would contain the application server a database, . . . , as architectural components.

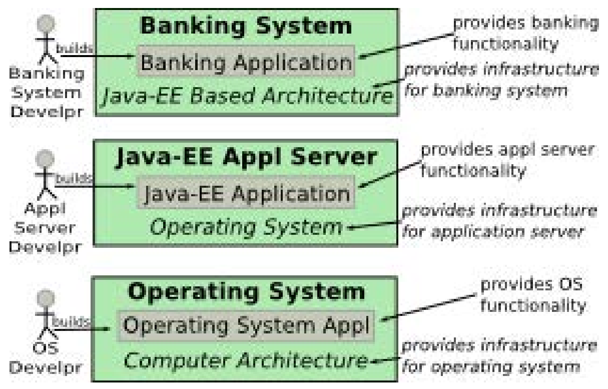
Banking System
Banking Application
Java-EE Based Architecture
builds
Banking System Develpr
provides banking functionality
provides infrastructure for banking system
Java-EE Appl Server
Java-EE Application
Operating System
builds
Appl Server Develpr
provides appl server functionality
provides infrastructure for application server
Operating System
Operating System Appl
Computer Architecture
builds
OS Develpr
provides OS functionality
provides infrastructure for operating system

Figure 4: Architecture is a matter of perspective.

If, on the other hand, one is busy developing an application server, then the application is the application server itself. It realizes the functional requirements of the application server users (e.g. the business system developers and system administrators).

The point of this section is to highlight that software architecture is not an absolute concept. The concept of a software architecture is relative to the application one is developing.

From the perspective of the banking system developer, the banking application needs to provide the functionality required by the banking system users, whilst the application server would be a core architectural component - it does not provide any banking functionality. On the other hand, for the application server developer, the application server application needs to address the functional requirements for the application server itself which may include functionality like load balancing, role based authorization, thread and object pooling and so on. This functionality provided by the application server is the functionality which is required to address non-functional requirements in the banking system domain.

The same piece of software, the application server in our case, may thus be part of the architecture from the perspective of some developers and the actual application providing application logic for others. These developers are, however, developing applications from different domains. In our case an application for the business domain and an application for a high-level technical domain.

3. CURRENT DEFINITIONS OF SOFTWARE ARCHITECTURE

In this section we group commonly used definitions of software architecture into three classes. We analyze the differences between these classes and critically assess them from the perspective of whether and how they are able to define the boundary between architecture and application functionality.

3.1 Classes of software architecture definitions

The first class of software architecture definitions defines software architecture as **a high-level abstraction of a software system** [30, 28, 14, 23]. For example, Zhang & Goddard define software architecture as a *high level abstraction representing the structure, behaviour, and non-functional properties of software systems*[30]. Viewing architecture as a high-level system abstraction raises a number of questions. For example, what exactly is high-level and where is the boundary between architecture and application functionality? If one focuses on a particular component of a software system, is there still architecture. How does one know? Furthermore, since in such a definition architecture needs to address both, high-level functional and non-functional requirements, would one not need the same requirements elements for architecture and application functionality and where is the boundary between these? How are non-functional requirements at lower levels of granularity addressed if there is no architecture at these lower levels of granularity? Note also that this class of definitions identifies high-level abstractions of different aspects of a software system, e.g. high-level structural and functional abstractions. This prompts Baragry and Reed to question whether there is a single concept for software architecture [6]. The different abstractions are usually modeled using either different views onto a single architectural model as in, for example, the Kruchten 4+1 View model of a software architecture [18], or by using different architectural models as is supported by the IEEE *Recommended Practice for Architectural Description of Software-Intensive Systems* [1].

The second class of software architecture definitions views software architecture as the **structure and externally visible properties of a software system** [7, 10, 1]. This class of definitions does not include functionality in the definition of software architecture and keeps the question on whether architecture is only high-level open. The focus is on components and relationships between the components themselves as well as relationships between the components and the system environment. One such commonly referenced definition is that provided by Bass, Clements and Katzman [7] which defines software architecture as the *structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.* Even though these definitions make it more feasible to separate architectural and functional aspects of a system, they still raise a number of questions. For example, what would the components be? Would they provide application functionality (e.g. an account management system) or should one include only components addressing more abstract infrastructural or technical concerns (e.g. a services bus). The above definition speaks of software elements in general and does not distinguish between architectural and application elements. It caters for quality attributes of a software system through the *externally visible properties.* But externally visible components can also refer to aspects of the system related to its functionality. The boundary between architectural and application functionality is thus again not clearly specified.

The UML 1.3 definition of software architecture also falls within the class of structure-based software architecture definitions. But unlike other definitions, it explicitly states that software architecture needs to be specified across levels of granularity. UML 1.3 defines the architecture of a system as *the organizational structure of a system. An architecture can be recursively decomposed into parts that interact* *through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems*[8]. Even though such a definition acknowledges the existence of software architecture across levels of granularity, it still does not provide a any guidelines on whether one should or how one would differentiate between architectural components and application classes and interfaces required for application functionality.

The third class of software architecture definitions places the focus on an architecture introducing **fundamental concepts and constraints within which the software system is to be designed and developed**[13, 28]. The most widely referenced example of this class of software architecture definitions is the IEEE specification of a *Recommended Practice for Architectural Description of Software-intensive Systems*[13]. The latest version, ISO/IEC/IEEE 42010, defines a software architecture as the *fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution* [13]. The fundamental concepts cater for concepts within which application logic can be defined (e.g. a service and a services contract in a service-oriented architecture). The fundamental properties generally refer to quality attributes of a system. Finally, fundamental principles of the system design and evolution can be used to refer to core design constraints. These are often specified as architectural styles. Examples of architectural styles include *pipes and filters*, *layering*, *blackboard*, *microkernel*, *hierarchical*, and others. The above definition also refers to system elements without distinguishing between architectural and application or functional elements, leaving once again the boundary between architecture and application design open for interpretation.

In summary, we have three classes of software architecture definitions which view a software architecture as (1) a high level abstraction of a software system, (2) the structure of a software system or (3) the fundamental concepts, properties and principles of design of a software system. Note that there is significant overlap across these classes of software architecture definitions. All definitions include system components and relationships between them as part of the definition. However, the focus of each of these classes of definitions is fundamentally different. Also, the definitions differ on whether software architecture

- is a high level infrastructure or there is architecture across levels of granularity,
- focuses on components and relationships or whether it is more concerned with introducing fundamental concepts and constraints for the system,
- includes the specification of behaviour and functionality, and
- explicitly includes the specification of ways in which non-functional requirements are to be addressed.

3.2 Relation to ADLs

It is well known that *Architecture Description Languages* (ADLs) differ extensively in what aspects of a software architecture they are able to describe [24, 17, 30]. Here we point out that features of the different classes of software architecture definitions are supported by different ADLs.

The commonality across the classes of software architecture definitions is reflected in virtually all ADLs having the ability to specify components and relationships between them. For example, *Wright* [2], *Acme* [16], and their derivatives are all based on a core of architectural components and relationships.

The first class of architecture definitions includes the specification of functionality as part of the architecture. Many ADLs including *Wright* [2], *Darwin* [20], *MADL* [27], and *CBabel* [9] include the ability to specify behaviour. This is typically done using formal process specification technologies like π-calculus or *Concurrent Sequential Processes* (CSP).

The second class of software architecture definitions includes externally visible properties as part of the definition. These are commonly seen as quality attributes of the system. Most ADLs do not provide explicit support for specifying quality attributes or strategies through which these are addressed. A notable exception is the aspect-oriented ADLs [25, 26] which use aspects to apply architectural strategies to address quality attributes. Another approach is that followed by Menase et al. [22]. They define a set of what they call architectural patterns representing concrete designs of architectural strategies. They show that components implementing these patterns can not only be statically selected in the software architecture specification, but can also be dynamically selected by self-adapting software architectures.

The third class of software architecture definitions puts emphasis on the definition of fundamental concepts and principles of design. These are commonly represented by architectural styles which introduce style-specific concepts (e.g. layers, pipes and filters, . . .) and define constraints on how these design elements may be assembled. Some ADLs, notably *Acme* [16], have explicit support for specifying architectural styles and constraints.

4. WHAT IS SOFTWARE ARCHITECTURE?

Based on the insights obtained from reference architectures we will introduce a simple definition of software architecture, discuss its elements and show that these are present across levels of granularity. Finally we discuss the interplay between architecture and functional design.

4.1 Definition of software architecture

We propose the following very simple definition for software architecture:

> Software architecture is the software infrastructure within which application components providing user functionality can be specified, deployed and executed.

The definition does rely on the definition of an *application component* [3] which we define as follows:

> Application components are software components which address functional requirements of the software system.

4.2 Elements of software architecture

In our analysis of reference architectures we have identified the concerns addressed by these reference architectures. From this we identify the elements of software architecture as

- basic concepts and constraints within which application components providing user functionality are specified,
- a set of architectural components addressing technical concerns,
- the integration channels to the environment as well as the internal integration infrastructure for both, architectural and application components, and
- a set of architectural strategies which are used to concretely address the quality requirements for the software system.

4.3 Boundary between architecture and application design

Our definition does not restrict architecture to a high-level abstraction of the system, but to an environment within which user functionality is deployed and executed. A system element is either part of the architecture or part of an application depending on whether it contains logic which addresses functional requirements from the problem/application domain or logic implementing architectural functionality addressing technical concerns.

4.4 Architecture and application components across levels of granularity

Our definition of software architecture does not refer to high-level infrastructure. Software architecture is the full software infrastructure, across levels of granularity, within which the user functionality is deployed. There is thus both, functional and architecture design, across levels of granularity.

From the perspective of the definition of software architecture proposed in this paper, there are this architectural and application components across levels of granularity with the former addressing infrastructural and quality concerns and the latter addressing concerns around application functionality.

In the analysis of reference architectures it was shown that all architectural elements may exist across levels of granularity. This includes architectural patterns, concepts, components, relationships and strategies.

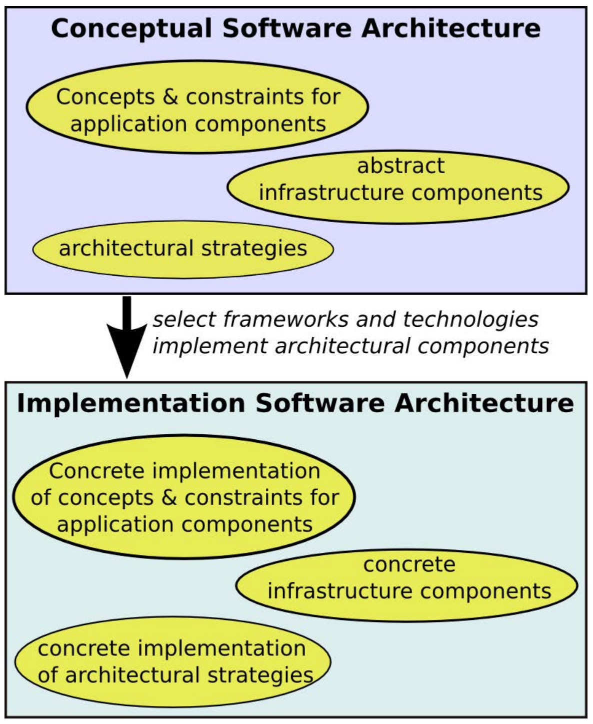
Conceptual Software Architecture
Concepts & constraints for application components
abstract infrastructure components
architectural strategies
select frameworks and technologies
implement architectural components
Implementation Software Architecture
Concrete implementation of concepts & constraints for application components
concrete infrastructure components
concrete implementation of architectural strategies

Figure 5: Conceptual versus implementation architecture.

4.5 Conceptual versus implementation architecture

One commonly differentiates between the conceptual architecture and the implementation architecture. The conceptual architecture may introduce concepts and constraints within which applications are developed, abstract architectural components addressing infrastructural concerns, architectural strategies which are to be used to address non-functional requirements.

The implementation architecture may provide a concrete infrastructure for application components, enforce some of the application component constraints, provide concrete implementations of the abstract architectural component of the conceptual architecture and concrete implementations of the architectural strategies.

4.6 Contextualization within MDE

In the context of model-driven engineering (MDE) the computation independent model (CIM) would be purely in the problem (e.g. business) domain, independent of the software architecture.

The platform independent model (PIM) would be a mapping of the CIM within the infrastructure as specified by the conceptual architecture, using the application concepts of the application architecture.

The PIM would then be mapped onto a platform specific model (PSM) which is specific to the implementation architecture and ultimately onto an implementation within code and other artifacts.

4.7 Interplay between infrastructural and functional design

So far we have said that anything which does not address the functional requirements of the problem domain is part of architecture. However, whether a particular component is an architectural or a functional component depends on one's perspective.

For example, whilst implementing, say, an online retail application which is to be deployed in an application server, the application server is part of the architecture. It contains no logic about retailing and solely addresses non-functional requirements like integration requirements and providing a deployment and execution infrastructure within which the reliability, scalability and security requirements of the retail application are addressed.

If however, ones perspective changes from developing the retail application to developing the application server itself, then the application is the application server itself. It will have to implement functional requirements around clustering, persistence and managing thread pools.

The application server, in turn, would have to be deployed and executed in an environment which addresses its non-functional requirements. The choice of suitable architecture would be made by selecting a suitable operating system.

Thus, even though a software architecture does not realize the functional requirements from the application domain, it has its own functional requirements which would themselves have to be addressed through functional design.

5. RELATED WORK

We have a variety of definitions of software architecture. These definitions have been abstracted into software architecture classes which have been discussed in section 3.

Baragry and Reef [6] discuss reasons why is is difficult to define software architecture. They compare software architecture with other disciplines of architecture in order to highlight the differences in the concept of architecture across different domains. Based on these differences, they question whether it makes sense to base the concept of software architecture on the analog of buildings architecture. The authors point out that the high-level abstractions of different aspects of a software system including those of its structure and functionality have to be specified in different representations of software architecture and not merely different views of a single concept. Comparing these different representations to the general notion of architecture, they conclude that each of these separate representations could be seen as the software architecture itself. The definition of software architecture provided in this paper does not define architecture as a high level abstraction but as an infrastructure within which application logic is deployed and executed. This is indeed a single concept.

A number of authors have previously focused on separating architectural concerns from application logic. This is similar to the approach taken in this paper. Deconinck, De Florio and Botti [11] focused on separating architectural concerns like reliability and maintainability form application logic. They address these concerns in a separate framework within which the application logic is deployed and discuss the benefits of separating these concerns. Dlodlo and Bamford [12] looked at how to separate the human adapter (user interface) from the application functionality. None of these papers provide a definition or a detailed discussion of the elements of software architecture.

Alexander Ran [28] highlights that a software architecture introduces a set of fundamental concepts used to define a system. He defines software architecture as *a set of concepts and design decisions about structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant, explicit functional and quality requirements, and implicit requirements of the problem and the solution domains.* The definition is, however, to some extent a circular definition as it relies on the concept of "architecturally significant" which in turn depends on the definition of architecture itself.

6. CONCLUSIONS AND OUTLOOK

Different classes of definitions of software architecture have been identified. These classes differ on whether architecture is a high level abstraction of the various system aspects (e.g. its structure and functionality), whether it is the system structure and externally visible properties across levels of granularity or whether the focus is on introducing concepts and constraints for the application development. It is not clear from either of these definition classes where exactly the boundary between architecture and functional design is. This uncertainty of architecture has significant implications for architecture requirements and descriptions, as well as architecture design and assessment approaches.

It has been shown that reference architectures are relatively pure software architecture specifications and implementing frameworks as relatively pure software architecture implementations. These reference architectures provide an infrastructure within which application functionality is deployed and executed. These reference architectures are consistent with defining a software architecture as *the software infrastructure within which application components providing user functionality can be specified, deployed and executed.* A software architecture specification based on the above definition of software architecture introduces basic concepts and constraints within which application components are to be specified, architectural components addressing infrastructural and quality concerns (including integration channels), and architectural strategies which are to be used to concretely address quality requirements. Both, application and architecture design needs to be done across levels of granularity, i.e. there are architectural components addressing infrastructural and quality concerns as well as application components providing user functionality across levels of granularity.

It has been pointed out that whether a system component is an architectural or an application component depends on one's perspective. In particular, one may develop a framework implementing a software architecture specification, addressing the non-functional requirements for the applications to be defined, deployed and executed. Such a framework, which would be an architectural component of the "business" application will, once again, have functional and non-functional requirements. The former are addressed through application design and the latter by selecting an appropriate architecture within which that architectural component can be deployed and executed.

Future work will focus on whether current architecture description languages have sufficient semantics to describe a software architecture and on defining an architecture analysis and design methodology which can be used in conjunction with the URDAD methodology[15, 29] for technology and architecture neutral analysis and design. In addition, this work opens opportunities for revisiting architecture analysis methods as well as architecture design methods.

7. ACKNOWLEDGEMENTS

The author would like to thank Stefan Gruner for valuable discussions and guidance and the University of Pretoria for funding the research.

8. REFERENCES

[1] IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. Specification, IEEE-SA Standards Board, Sept. 2000.

[2] R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon, School of Computer Science, Jan. 1997. Issued as CMU Technical Report CMU-CS-97-144.

[3] A. Ambroziewicz and M. Smialek. Application logic patterns reusable elements of user-system interaction. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 241–255, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] P. Avgeriou. Run-time Reconfiguration of Service-Centric Systems. In *Proceedings of the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)*, Irsee, Germany, 2006.

[5] F. Bachmann, L. Bass, M. Klein, and C. Shelton. Designing Software Architectures to Achieve Quality Attribute Requirements. *IEE Proceedings - Software*, 152(4):153–165, Aug. 2005.

[6] J. Baragry and K. Reed. Why is it so hard to define software architecture? In *Software Engineering Conference, 1998. Proceedings. 1998 Asia Pacific*, pages 28 –36, dec 1998.

[7] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition.* Addison-Wesley Professional, Apr. 2003.

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[9] C. Bouanaka and F. Belala. Towards a mobile architecture description language. In *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*, pages 743 –748, Apr. 2008.

[10] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley Professional, 2 edition, Oct. 2010.

[11] G. Deconinck, V. De Florio, and O. Botti. Separating recovery strategies from application functionality: experiences with a framework approach. In *Reliability and Maintainability Symposium, 2001. Proceedings. Annual*, pages 246 –251, 2001.

[12] N. Dlodlo and C. Bamford. Separating application functionality from the user interface in a distributed environment. In *EUROMICRO 96. 'Beyond 2000: Hardware and Software Design Strategies'., Proceedings of the 22nd EUROMICRO Conference*, pages 248 –255, sep 1996.

[13] D. Emery and R. Hilliard. Every Architecture Description Needs a Framework: Expressing Architecture Frameworks Using ISO/IEC 42010. In *Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009*, pages 31–40. IEEE, Sept. 2009.

[14] M. Fowler. Design - who needs an architect? *Software, IEEE*, 20(5):11 –13, 2003.

[15] Fritz Solms and Dawid Loubser. URDAD as a semi-formal approach to analysis and design. *Innovations in Systems and Software Engineering*, 6:155–162, 2010. 10.1007/s11334-009-0113-4.

[16] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of component-based systems*, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.

[17] S. Giesecke, J. Bornhold, and W. Hasselbring. Middleware-Induced Architectural Style Modelling for Architecture Exploration. In *The Working IEEE/IFIP Conference on Software Architecture, 2007. WICSA '07*, pages 21–21. IEEE, Jan. 2007.

[18] P. Kruchten. The 4+1 view model of architecture. *Software, IEEE*, 12(6):42 –50, nov 1995.

[19] P. T. L. Lloyd and G. M. Galambos. Technical reference architectures. *IBM Systems Journal*, 38(1):51–75, 1999.

[20] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Software Engineering—ESEC'95*, pages 137–153, 1995.

[21] S. Malek, N. Esfahani, D. A. Menasce, J. P. Sousa, and H. Gomaa. Self-Architecting Software SYstems (SASSY) from QoS-annotated activity models. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, PESOS '09, pages 62–69, Washington, DC, USA, 2009. IEEE Computer Society.

[22] D. A. Menasce, J. P. Sousa, S. Malek, and H. Gomaa. Qos architectural patterns for self-architecting software systems. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 195–204, New York, NY, USA, 2010. ACM.

[23] M. Nagl. *Software Engineering – Methodological Programming-in-the-Large (in German).* Springer-Verlag, 1990.

[24] C. Pahl, S. Giesecke, and W. Hasselbring. An Ontology-Based Approach for Modelling Architectural Styles. In F. Oquendo, editor, *Software Architecture*, volume 4758, pages 60–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[25] M. Pinto and L. Fuentes. AO-ADL: An ADL for Describing Aspect-Oriented Architectures. In A. Moreira and J. Grundy, editors, *Early Aspects: Current Challenges and Future Directions*, volume 4765, pages 94–114. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[26] M. Pinto, L. Fuentes, and J. M. Troya. Specifying aspect-oriented architectures in AO-ADL. *Information and Software Technology*, 53(11):1165 – 1182, 2011. <ce:title>AMOST 2010</ce:title> <ce:subtitle>AMOST 2010</ce:subtitle>.

[27] W. Qin and S. Malik. A Study of Architecture Description Languages from a Model-based Perspective. In *Microprocessor Test and Verification, 2005. MTV '05. Sixth International Workshop on*, pages 3 –11, Nov. 2005.

[28] A. Ran. Fundamental concepts for practical software architecture. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 328–329, New York, NY, USA, 2001. ACM.

[29] F. Solms, C. Edwards, A. Paar, and S. Gruner. A Domain-Specific Language for URDAD Based Requirements Elicitation. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment*, SAICSIT '11, pages 224–230, New York, NY, USA, 2011. ACM.

[30] S. Zhang and S. Goddard. xSADL: an architecture description language to specify component-based systems. In *International Conference on Information Technology: Coding and Computing, 2005. ITCC 2005*, volume 2, pages 443– 448 Vol. 2. IEEE, Apr. 2005.

[31] L. Zhu, M. Staples, and V. Tosic. On Creating Industry-Wide Reference Architectures. In *12th International IEEE Enterprise Distributed Object Computing Conference, 2008. EDOC '08*, pages 24–30. IEEE, Sept. 2008.