Introduction to software architecture

Introduction *7*

1 What is software architecture? *8*
- 1.1 A definition *8*
- 1.2 Architecture in the life cycle *8*
- 1.3 Why do we need software architecture? *9*
- 1.4 Software architecture as a discipline *11*
- 1.5 Scope and focus of architecture *11*

2 Topics in software architecture *12*
- 2.1 Stakeholders, concerns and requirements *12*
- 2.2 The ISO 25010 quality framework *13*
- 2.3 Describing architectures *14*
- 2.4 Architecture and reuse *16*

3 The architect versus the engineer *19*

Discussion questions *19*

CC BY NC ND

Learning unit 1

Introduction to software architecture

INTRODUCTION

The subject of this course is software architecture. In this first learning unit, we will explain what software architecture is. You will find out why we need software architecture and you will see aspects of scientific research that is done on software architecture. You will be introduced to the several topics within the field of software architecture that will be covered in this course: software architecture as a solution that balances the concerns of different stakeholders, quality assurance, methods to describe and evaluate architectures, the influence of architecture on reuse, and the life cycle of a system and its architecture. This learning unit concludes with a comparison between the professions of software architect and software engineer.

All aspects of software architecture that are introduced in this learning unit will be treated in more detail in individual learning units.

LEARNING GOALS
After having studied this learning unit, you will be expected to be able to:
- describe the place of software architecture within the life cycle
- explain the need for an architecture
- describe the responsibilities of a software architect
- explain the relationship between stakeholders and a system's architecture
- describe the role of requirements in software architecture
- explain the role of compromise in creating an architecture
- explain the relationship between architecture and reuse

Study advice
The reading assignments for the textbook and the reader, for all course units, can be found on the course website.

The course website also provides an indication of the number of hours which, in our estimation, you will need for each course unit.

CC BY NC ND

LEARNING CORE

1 What is software architecture?

1.1 A DEFINITION

Definition 1 (Software architecture) IEEE *standard 1471 defines* software architecture *as the fundamental organisation of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.*

An architecture embodies information about components and their interaction, but omits information about components that does not pertain to their interaction. Thus, an architecture is foremost an abstraction of a system that suppresses details of components that do not affect the use, the relations and interactions of components. Private details of components, details that have to do solely with internal implementation and are not externally visible, are not architectural. In short, an architecture determines how components interact, not how they are implemented.

Every system has an architecture, but it does not follow that the architecture is known to anyone. Perhaps the designers of the system are long gone, perhaps documentation was never produced or perhaps the source code has been lost.

This shows that an architecture is not the same as a description of an architecture. While most learning units concern different aspects of architectures, one of the learning units is explicitly dedicated to the description of architectures.

1.2 ARCHITECTURE IN THE LIFE CYCLE

Figure 1.1 shows the life cycle of a system and the place of architecture in the life cycle. Software architecture links requirements analysis with realisation, and it may be explored and defined incrementally. Its existence allows one to predict quality aspects before realisation. The architecture must be stable before major development starts.

An architecture can help evolutionary development, when evolution is one of the requirements. A software architecture may be seen as a blueprint and guideline for realisation.

Definition 2 (Architectural conformance) Architectural conformance *is the extent to which the architecture is actually used.*

Conformance may be enforced by management. For optimal conformance, an architecture should be well documented. An architecture

CC BY NC ND

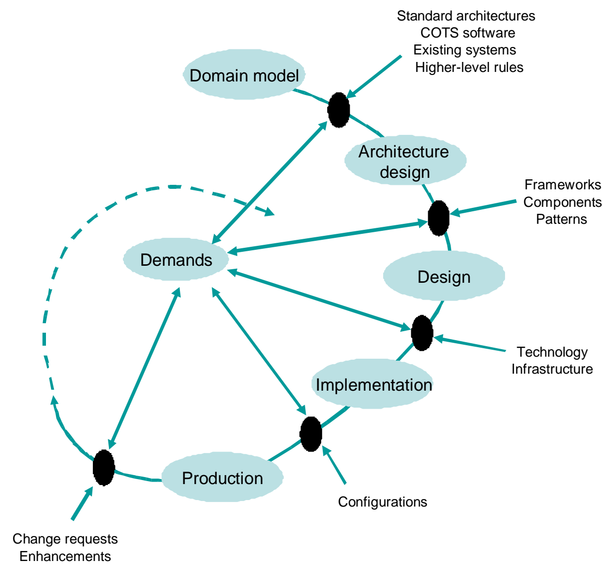

FIGURE 1.1 System life cycle

should also state clear principles, for instance, 'no database access from the servlet layer'. Architectural conformance should also be maintained over time.

Architectural decay

Architectural decay is the opposite of architectural conformance over time: it is often the case that software drifts from the original architecture through maintenance and change operations.

1.3 WHY DO WE NEED SOFTWARE ARCHITECTURE?

Applications are becoming larger and more integrated and are implemented using a wide variety of technologies. The various technologies and disciplines need to be orchestrated to ensure product quality. Quality attributes like reliability or usability cannot be analysed at the code level, but they can be analysed at the software architectural level.

Software architecture has several functions, which we describe below.

1.3.1 *Software architecture as a means for communication*

In the first place, we need an architecture as a means for communication among stakeholders.

Definition 3 (Stakeholder) *A* stakeholder *is anyone with a legitimate interest in the construction of a software system. Stakeholders include the customer, the end users, the developers, project management and the maintainers, among others. Stakeholders are representatives from all stages of development, usage and support.*

CC BY NC ND

Customers, users, developers as well as maintainers, all are considered stakeholders. A software architecture represents a common high-level abstraction of a system that can be used by all of the system's stakeholders as a basis for creating mutual understanding, forming consensus and communicating with each other [7].

1.3.2 *Software architecture as a representation of early design decisions*

Early design decisions

A second function of software architecture is that it is a representation of *early design decisions*. A software architecture is the documentation of the earliest design decisions about a system, and these early decisions carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its deployment and its maintenance life. It is also the earliest point at which the system to be built can be analysed.

1.3.3 *Software architecture as a basis for a work-breakdown structure*

Work-breakdown

Software architecture does not only prescribe the structure of the system being developed. That structure also becomes engraved in the structure of the development project. Because the architecture includes the highest-level decomposition of the system, it is typically used as the basis for the *work-breakdown* structure. This dictates units of planning, scheduling and budget, and effectively freezes the architecture. Development groups typically resent relinquishing responsibilities, which means that it is very difficult to change the architecture once these units have begun their work.

1.3.4 *Software architecture as a means to evaluate quality attributes*

It is possible to determine whether the appropriate architectural choices have been made (i.e. that the system will exhibit its required quality attributes) before the system is fully developed and deployed, as software architecture allows one to predict system qualities. This is further explained in the learning unit dedicated to architecture evaluation.

1.3.5 *Software architecture as a unit of reuse*

Another function of software architecture is that it is a transferable abstraction of a system. A software architecture constitutes a relatively small, intellectually graspable model of the structure of a system and the way in which its components work together. This model is transferable across systems. In particular, it can be applied to other systems exhibiting similar requirements and can promote large-scale reuse.

Software product line

An example of how software architecture promotes large-scale reuse is the construction of product lines. A *software product line* is a family of software systems that shares a common architecture. The product line approach is a way to gain quality and save labour.

cc BY NC ND

Another example of how software architecture promotes large-scale reuse is component-based development. Software architecture complements component-based and services-based development.

Many software engineering methods focus on programming as the prime activity, with the progress measured in lines of code. Architecture-based development often focuses on composing or assembling components that are likely to have been developed separately or even independently from each other.

1.4 SOFTWARE ARCHITECTURE AS A DISCIPLINE

Mary Shaw made software architecture into a discipline with her 1989 article 'Larger Scale Systems Require Higher Level Abstractions' [55]. She showed that organising systems on the subsystem and module level is different from organising code.

The field of software architecture has taken inspiration from other engineering domains, such as architecture and electronics. The concepts of stakeholders and concerns, analysis and validation, styles and views, standardisation and reuse, best practices and certification are all well-known in these domains. However, software is different from the products in all other engineering disciplines. Rather than delivering a final product, delivery of software means delivering a blueprint for products. Computers can be seen as fully automatic factories that accept such blueprints and instantiate them.

The main consequence is that plans can be parameterised, applied recursively, scaled and instantiated any number of times. There are no reproduction costs. Unfortunately, invisible complexity may lead to unreasonable assumptions, for instance about flexibility.

1.5 SCOPE AND FOCUS OF ARCHITECTURE

Scope and focus are two dimensions of software architectures.

Definition 4 (Scope) *The* scope *of a software architecture concerns the range of applications to which it pertains.*

An architecture describes at least a single system or product, such as a specific version of Microsoft Word. A more general scope is that of a system family. An example of a system family is the software for medical imaging used by companies like Philips Medical Systems. This family consists of various products that produce images on various media (computer monitor, photographic film) by means of various techniques (X-ray, ultrasound, MRI scan), but all of these products are built along similar architectural lines. The scope may be that of a business unit, of an organisation or enterprise as a whole or of a domain, or the scope may be generic.

An example of a domain architecture is a compiler. A compiler generally consists of several basic elements (the front end, back end, symbol table and such) that behave in a well-known way and are intercon-

CC BY NC ND

nected in a regular fashion. A person designing a compiler would not start from scratch, but would begin with this basic domain architecture in mind when defining the software architecture of this new compiler.

Focus

Another aspect in which architectures differ is their *focus*:

Application architecture

- An *application architecture* is a blueprint for individual applications, their interactions and their relationships to the business processes of the organisation. It is built on top of the IT architecture.

Information architecture

- *Information architecture* is concerned with logical and physical data assets and data management resources.

IT architecture

- An IT *architecture* defines the hardware and software building blocks that make up the overall information system of the organisation. The business architecture is mapped to the IT architecture. The purpose of an IT architecture is to enable the company to manage its IT investment in a way that meets its business needs. It includes hardware and software infrastructure, including database and middleware technologies.

Business architecture

- A *business architecture* defines the business strategy, governance, organisation and key business processes within an enterprise. The field of business process reengineering (BPR) focuses on the analysis and design of business processes; these are not necessarily represented in an IT system.

2 Topics in software architecture

Topics within the field of software architecture are:

- **definition of the solution structure**. The solution structure is defined using concepts such as components and connectors, contracts, frameworks and services.
- **quality assurance**, in relation to stakeholders and their concerns. A rationale explains how a certain decision is related to concerns of stakeholders. You can use analysis, validation or assessment for this.
- **describing architectures**. Relevant concepts are viewpoints, models and views. You can use languages, notations and visualisation.
- **architecture and reuse**. Relevant concepts are reusable architectures, architectural styles and patterns. Specification and connection of components, standardisation, commercial off-the-shelf (COTS) components, product lines, redesign of legacy systems and service-oriented architectures are all related to reuse.
- **architecture in the life cycle** of a system: a methodology for maintenance and evolution of the architecture is required.

We will introduce these topics in this learning unit; you will read about them in more detail in later learning units.

2.1 STAKEHOLDERS, CONCERNS AND REQUIREMENTS

Stakeholders

Customers, end users, developers, the developer's organisation and those who maintain the system are all examples of *stakeholders*. An architecture must balance the concerns of the different stakeholders.

CC BY NC ND

All stakeholders have their own concerns. Customers, for example, want a system that is easy to understand and performs well on a particular piece of hardware. Marketing people want to achieve a short time to market, a low cost of development and an easily customised system.

Definition 5 (Concern) *A* concern *is an interest related to the development, operation, use or evolution of a system. A concern may be related to functionality, quality areas, costs, technology and so forth. A concern is expressed from the point of view of a stakeholder.*

Concerns are translated into requirements. The following are examples of requirements: providing a certain behaviour at run-time, performing well on a particular piece of hardware, being easy to customise, achieving a short time to market, low cost of development and gainfully employing programmers who have a particular specialty. Requirements are preferably measurable.

Requirements are very important: it is costly or impossible to repair or ignore them. Requirements may be contradictory. Speed, flexibility and reliability for instance, may ask for different solutions. It is therefore important to prioritise.

The trade-offs between performance and security, between maintainability and reliability and between the cost of initial development and the cost of future developments, are all manifested in the architecture. The system's behaviour on these quality issues is the result of structural trade-offs made by the developers, and is traditionally undocumented!

Architects must identify and actively engage the stakeholders to solicit their needs and expectations. Without such active engagement, the stakeholders will, at some point, explain to the architects why a proposed architecture is unacceptable, thus delaying the project. Early engagement allows architects to understand the constraints of the task, manage expectations and negotiate priorities.

The architect must understand the nature, source, and priority of these constraints. This places the architect in a better position to make trade-offs when conflicts among competing constraints arise, as they inevitably will.

The architect needs more than just technical skills. Stakeholders will have to be informed on a continuing basis of why priorities have been chosen and why some expectations cannot be met. Diplomacy, negotiation and communications skills are essential.

2.2 THE ISO 25010 QUALITY FRAMEWORK

Definition 6 (Quality framework) *A* quality framework *establishes a terminology for quality attributes (as a common language for negotiation with and among stakeholders) and also establishes measures for these attributes.*

CC BY NC ND

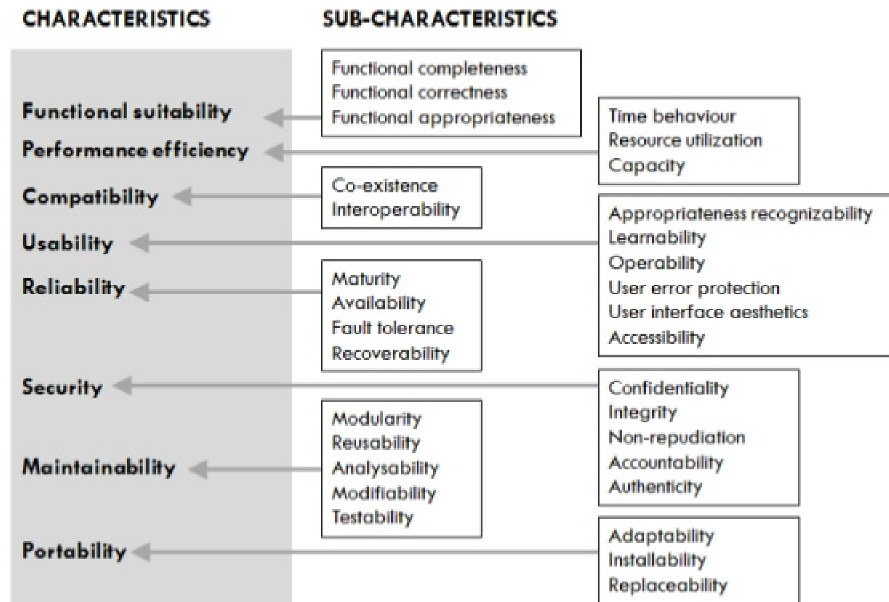

FIGURE 1.2 Parameters of quality

Figure 1.2 presents the properties that are used as parameters of quality. Quality must be considered at all phases of design, implementation and deployment, but different qualities manifest themselves differently during these phases. For instance, many aspects of usability are not architectural: making the user interface clear and easy to use is primarily a matter of getting the details correct (radio button or checkbox?). These details matter tremendously to the end user, but they are not architectural.

The quality properties that are important to the requirements are assessed on the basis of the description of an architecture. When analysis of the description shows that the desired properties will not be fulfilled, improvements to the architecture are needed. Properties cannot always be automatically analysed. Sometimes, they cannot be analysed at all or simply cannot be analysed at this early stage of system design. Therefore, reviews by experts are also an important method for quality assurance.

Quality assurance

The following are techniques for *quality assurance*: applying specific measures to design, technology, process and so on, building architectural prototypes, reviews by experts, assessment methods and methods for automatic analysis (which require suitable models and descriptions).

2.3 DESCRIBING ARCHITECTURES

Viewpoint

Quality assessment of architectural decisions requires descriptions of architectures. An important concept in architecture description is the notion of *viewpoint*. Different stakeholders require descriptions of the architecture from different viewpoints: developers, maintainers, end users, project managers, service engineers, auditors and other architects are interested in different aspects of the architecture.

cc BY NC ND

There are several reference models for architectural descriptions that use different viewpoints, such as the Kruchten 4+1 model [34], the viewtypes of Clements et al. [10] or the viewpoints and perspectives of Rozanski and Woods [53].

Several languages are used to describe architectures. Natural languages and the Unified Modelling Language (UML) [46] are often used, as are pictures without a formal basis. More formal notations are also used: Module Interconnection Languages (MILs) and Architecture Description Languages (ADLs). These are names for classes of languages. Some of these languages will be discussed in the learning unit on describing and evaluating architectures.

Models

An architecture can be characterized as a set of *models*. We build models of complex systems because we cannot comprehend such a system in its entirety: a model is a simplification of reality. We build models to understand the system we are building. Models can be used for visualisation or specification, as a template for construction or for documenting decisions. Every model can be expressed at different levels of precision.

View

A *view* is a projection of a model, omitting entities that are irrelevant from a certain perspective or vantage point. No single model suffices. Systems are best approached through a small set of nearly independent models with multiple views.

Functional viewpoint

– Views belonging to the *functional viewpoint* describe the system's runtime functional elements and their responsibilities, interfaces and primary interactions. The functional view of a system defines the architectural elements that deliver the system's functionality. These views document the system's functional structure—including the key functional elements, their responsibilities, the interfaces they expose and the interactions between them. Diagrams used in these views model elements, connectors, interfaces, responsibilities and interactions, for instance using UML component diagrams.

Information viewpoint

– Views belonging to the *information viewpoint* describe the way in which the architecture stores, manipulates, manages and distributes information. This viewpoint concerns both the information structure and the information flow. DFD's (Data Flow Diagrams), UML Class diagrams, ER (entity-relation) diagrams and so on, for instance are used as diagrams for these views.

Concurrency viewpoint

– Views belonging to the *concurrency viewpoint* describe the concurrency structure of the system, mapping functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently, and to show how the model is coordinated and controlled. This viewpoint considers the systems concurrency and state-related structure and constraints. UML state diagrams or UML component diagrams, for instance, can be used as diagrams for these views.

Development viewpoint

– Views belonging to the *development viewpoint* describe the architecture that supports the software development process. Module organisation, standardisation of design and testing, instrumentation, code structure, dependencies and configuration management are aspects of this. UML component diagrams with packages, for instance, are used as diagrams for this view.

CC BY NC ND

Deployment viewpoint

– Views belong to the *deployment viewpoint* describe the environment into which the system will be deployed, including the dependencies the system has on its run-time environment. Specifications of required hardware, of required software or of network requirements are aspects of this viewpoint. UML deployment diagrams are used as diagrams for this view.

Operational viewpoint

– Views belonging to the *operational viewpoint* describe how the system will be operated, administered and supported when it is running in its production environment. Installation and upgrade, functional migration, data migration, monitoring and control, backup, configuration management and so on are aspects in this viewpoint.

Depending on the nature of the system, some views may be more important than others. In data-intensive systems, views addressing the information viewpoint will dominate. In systems which will have many short release cycles, the deployment viewpoint and the views belonging to it will be important. In real-time systems, the concurrency viewpoint will be important.

The choice of what models to create has a profound influence on how a problem is tackled and a solution is shaped. The models you choose greatly affect your worldview. If you build a system through the eyes of a database developer, you will likely focus on entity-relationship models. If you build a system through the eyes of a structured analyst, you will likely end up with models that are algorithm-centric, with data flowing from process to process. If you build a system through the eyes of an object-oriented developer, you will end up with a system with an architecture centred around classes and their interactions. Each worldview leads to a different kind of system, with different costs and benefits.

2.4 ARCHITECTURE AND REUSE

Architectures are reusable artefacts. Reuse comes in different forms. Architectural patterns and styles may be reused. Commercial frameworks and platforms with infrastructural support are another way of reuse. One example of a framework is the Struts framework for web applications using the Model-view-controller pattern [4]. The J2EE platform is an example of a platform [61].

To enable reuse, architecture standards are necessary, with clear component roles. These standards may be aimed at a specific domain, or at a specific technology.

2.4.1 *Product lines*

Product lines

Product lines form an example of reuse-driven development. Figure 1.3 shows the number of product variants from left to right and the number of sales per variant from below to above. It shows how the production of individual cars using traditional craftsmanship (Bugatti) has evolved into mass production of a single model (Ford Model T), and later to

BY NC ND

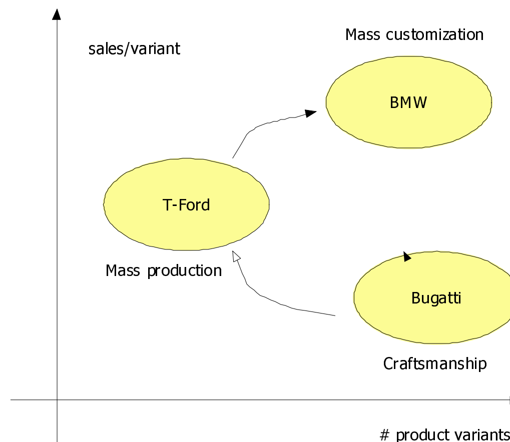

FIGURE 1.3 Towards product lines within the automotive industry

mass production and mass customisation, which makes it possible to produce multiple models in great quantities. Mass customisation in the automotive industry is comparable to product lines in the software industry.

Definition 7 (Software product line) *A* software product line (SPL) *is a set of software-intensive systems that share a common, managed set of functional modules and non-functional features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed manner [9].*

Software product line

Commonality
Variability

In other words, a software product line is a family of similar systems. The commonality of the systems is called the *commonality*; the variations are called the *variability*. Requirements for products within a product line are modelled in a feature model (where feature stands for a requirement from a user's perspective), which makes it possible to discern common features for all products within the product line and to discern features that vary among different products.

Common components or a framework offer the common features; the variability is offered by application-specific extensions. Domain engineering addresses the systematic creation of domain models and architectures, leading to common components or frameworks. Application engineering uses the models and architectures (and the common components and frameworks) to build systems.

In the words of the Carnegie Mellon Software Engineering Institute (SEI) [9]:

> **What is a Software Product Line?**
> A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features that satisfy the specific needs of a particular market segment or

CC BY NC ND

mission and that are developed from a common set of core assets in a prescribed manner.

Why are Software Product Lines Important?
Software product lines are rapidly emerging as a viable and important software development paradigm that allows companies to realise order-of-magnitude improvements in time to market, cost, productivity, quality and other business drivers. Software product line engineering can also enable rapid market entry and flexible response and can provide a capability for mass customisation.

2.4.2 *Reverse engineering*

Development often involves legacy systems that need to be reused, and sometimes need to be reorganised.

Definition 8 (Legacy system) *A* legacy system *is an existing computer system or application program which continues to be used because the user (typically an organisation) does not want to replace or redesign it.*

When we deal with a legacy system, we rarely have up-to-date documentation. The architecture models often need to be recovered, the data types transformed and the source code refactored.

Definition 9 (Reverse engineering) *The process of recovering the architecture of a legacy system is called* reverse engineering.

Reverse engineering
Re-engineering

Reverse engineering is a process of examination; the software system under consideration is not modified. Doing the latter would make it *re-engineering*.

Refactoring

Refactoring means modifying source code without changing its external behaviour. Refactoring does not fix bugs or add new functionality. The goal of refactoring is to improve the understandability of the code, change its structure and design, remove dead code or make it easier to maintain the code in the future.

An example of a trivial refactoring is changing a variable name into one that conveys more information, such as from a single character *'i'* to *'interestRate'*. A more complex refactoring is to turn code within an if-block into a subroutine.

Reverse engineering is based on either human cognition or technical approaches.
– Cognitive strategies are based on how humans understand software. A human might take a top-down approach to understanding software (start at the highest level of abstraction and recursively fill in

CC BY NC ND

understanding of the sub parts), a bottom-up approach (start by understanding the lowest-level components and how these work together), a model-based approach (if the investigator already has a mental model of how the system works and tries to deepen the understanding of certain areas) or an opportunistic approach (any combination of these approaches).

– Technical approaches centre on extracting information from the system's artefacts, including source code, comments, user documentation, executable models, system descriptions and so forth. Successful techniques of reverse engineering include extracting cross-reference information using compiler technology, data flow analysis, profiling to determine execution behaviour, natural language analysis and source code pattern matching at the architectural level.

3 The architect versus the engineer

Software architecture is a separate domain that requires professionals: software architects. Whereas an engineer analyses the precise requirements of a system, develops a detailed solution and ensures proper implementation and working of a system or a part of a system, a software architect creates a vision (an architecture), manages stakeholders' expectations and maintains the vision.

DISCUSSION QUESTIONS

a Discuss the role of software architecture in open source development.
b Do all stakeholder concerns carry equal weight? If not, what criteria exist for arbitrating among them?
c Where does the analogy between software architecture and building architecture break down?
d Discuss the role of software architecture with respect to agile methods such as Extreme Programming.
e Figure 1.4 shows an overview of the software architecture of the OpenOffice suite [47] as it is shown in the documentation. Discuss which viewpoint(s) this view belongs to.

CC BY NC ND

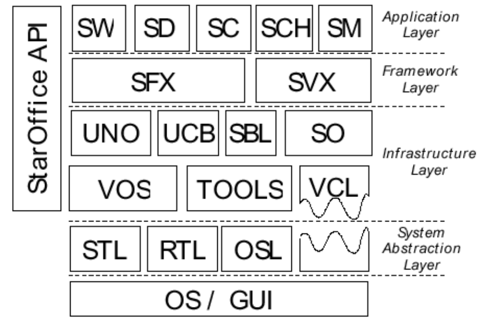

FIGURE 1.4 Architecture Overview