The 3+1 Approach to Software Architecture Description Using UML Revision 2.4

Henrik Bærbak Christensen, Aino Corry, and Klaus Marius Hansen
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{hbc,apaipi,marius}@daimi.au.dk

May 2016

Abstract

This document presents a practical way of describing software architectures using the Unified Modeling Language. The approach is based on a "3+1" structure in which three viewpoints on the described system are used – module, component & connector, and allocation – are used to describe a solution for a set of architectural requirements.

1 Introduction

Software architecture represents an appropriate level of abstraction for many system development activitites [Bass et al., 2003]. Consequently and correspondingly, appropriate software architectural descriptions may support, e.g., stakeholder communication, iterative and incremental architectural design, or evaluation of architectures [Bass et al., 2003], [Clements et al., 2002b], [Clements et al., 2002a].

This document represents a practical basis for architectural description and in doing so, we follow the IEEE recommended practice for architectural description of software-intensive systems [Software Engineering Standards Committee, 2000]. Central to this recommended practice is the concept of a *viewpoint* through which the software architecture of a system is described (see Figure 1). A concrete architectural description consists of a set of *views* corresponding to a chosen set of viewpoints. This document recommends the use of three viewpoints (in accordance with the recommendations of [Clements et al., 2002a]):

- A *Module viewpoint* concerned with how functionality of the system maps to static development units,
- a *Component & Connector viewpoint* concerned with the runtime mapping of functionality to components of the architecture, and
- an *Allocation viewpoint* concerned with how software entities are mapped to environmental entities

In addition to the views on the architecture, we recommend collecting architecturally significant requirements (see Section 2) in the architecture documentation. This corresponds to the *mission* of a system as described in [Software Engineering Standards Committee, 2000].

The views corresponding to these viewpoints are described using the Unified Modeling Language standard (UML; [OMG, 2003]). This reports provides examples of doing so. The UML has certain shortcomings in describing software architectures effectively[1], but is used here to strike a balance between precision/expressiveness and understandability of architectural descriptions.

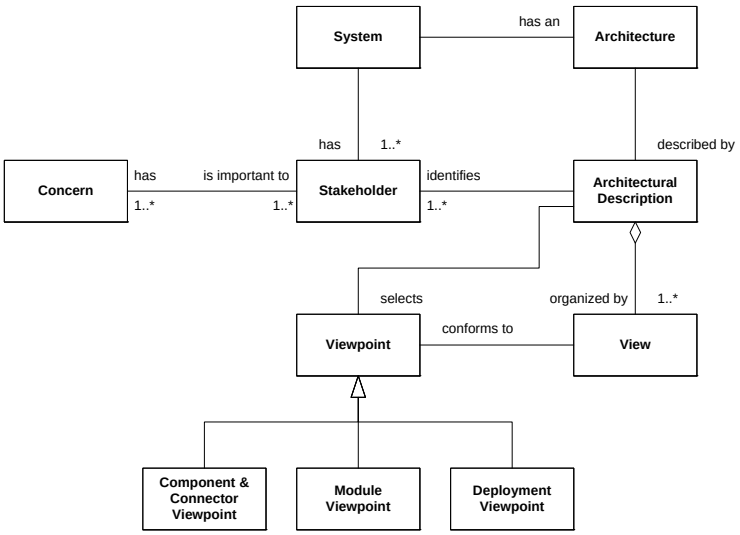

Figure 1: Ontology of architectural descriptions

1.1 Structure

The rest of this document is structured in two main sections: One introducing the "Architectural Requirements" section of the documentation (Section 2, page 2), and one introducing the "Architectural Description" section of the documentation (Section 2, page 2). These sections are introduced in general and a specific example of applying them to the documentation of a system is provided.

The examples are created to describe a point-of-sale system (NextGen POS) for, e.g., a supermarket point-of-sales. The example is inspired by the case study of Larman [Larman, 2002]. The system supports the recording of sales and handling of payments for a generic store; it includes hardware components as a bar code scanner, a display, a register, a terminal in the inventory hall, etc. More details of the functionality of the system can be found in Section 2.

[1]This is in particular connected to the central Component & Connector viewpoint [Clements et al., 2002a]

2 Architectural Requirements

Two types of descriptions of architecturally significant requirements are appropriate: scenario-based and quality attribute-based requirements.

The architecturally significant scenarios (or use cases) contain a subset of the overall scenarios providing the functional requirements for the system. These can possibly be augmented with requirements on performance, availability, reliability etc. related to the scenarios. Moreover, "non-functional" scenarios, e.g., describing modifiability of the system may be useful as a supplement[2].

All requirements cannot be described as scenarios of system functionality, and we propose supplementing the scenarios with a set of the most critical quality attributes that the system should fulfil. Since quality attributes (such as modifiability and performance) are often in conflict, this needs to be a subset of all architectural quality attributes.

The goal of describing architectural requirements is to enable the construction of a set of "test cases" against which different architectural designs may be compared and/or evaluated.

2.1 Example

In the NextGen POS case, a scenario is a specific path through a use case. An example of such a scenario is:

> *Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.*

Critical architectural attributes for the NextGen POS system are[3]:

- *Availability.* The system shall be highly available since the effectiveness of sales depends on its availability
- *Portability.* The system shall be portable to a range of different platforms to support a product line of POS systems
- *Usability.* The system shall be usable by clerks with a minimum of training and with a high degree of efficiency

3 Architectural Description

It is beneficial, when documenting software architecture, to apply different viewpoints to the system. Otherwise the description of the system will be incomprehensible.

[2] Architecturally significant scenarios are the basis of many architectural evalation approaches [Clements et al., 2002b]

[3] Note that this choice of quality attributes excludes, e.g., performance, scalability, security, safety, reliability, integrability, and testability.

Taken this into account, it is first important with a viewpoint which describes the functionality of the system in terms of how functionality is mapped into implementation. Secondly, it is important to describe how the functionality of the system maps to components and interaction among components. And thirdly, it is important to see how software components map onto the environment, in particular hardware structures. These three viewpoints are the module, component & connector, and allocation viewpoints respectively in concordance with [Clements et al., 2002a].

The viewpoints used in the architectural description section are defined as proposed in [Software Engineering Standards Committee, 2000]: for each, we first have a section describing the concerns of this viewpoint, then a section describing the stakeholders, then a section describing the elements and relations that can be used to describe views in this viewpoint, and finally an example of a view.

3.1 Module Viewpoint

3.1.1 Concerns

This architectural viewpoint is concerned with how the functionality is mapped to the units of implementation. It visualizes the static view of the systems architecture by showing the elements that comprise the system and their relationships.

3.1.2 Stakeholder Roles

This viewpoint is important to architects and developers working on or with the system.

3.1.3 Elements and Relations

The elements are units of implementation including:

> Class: A class describing the properties of the objects that exist at runtime.
>
> Package: A logical division of classes in the system. This can refer to packages as we find them in Java or just give a logical division between the classes of the system.
>
> Interface: A classification of the interface of the element that realizes it. It can refer to the interfaces found in e.g. Java or just a description of an interface that a class can conform to.

The relations describe constraints on the runtime relationships between elements:

> Association: Shows that there is a hard or weak aggregation relationship between the elements and can be used between classes.
>
> Generalization: Shows that there is a generalization relation between the elements and can be used between two classes or two interfaces.

Realization: Shows that one element realizes the other and can be used from a class to the interface it implements.

Dependency: Shows that there is a dependency between the elements and can be used between all the elements.

3.1.4 Examples

The module view of the POS system can be described using the class diagrams of UML, which can contain all the above mentioned elements and relations.

It is possible to describe the system top-down by starting with the most top-level diagram. In figure 2 the overall packages of the system are shown. Figure 3 and figure 4 show further decomposition of the Domain Model package and the Payments package in the Domain Model package.

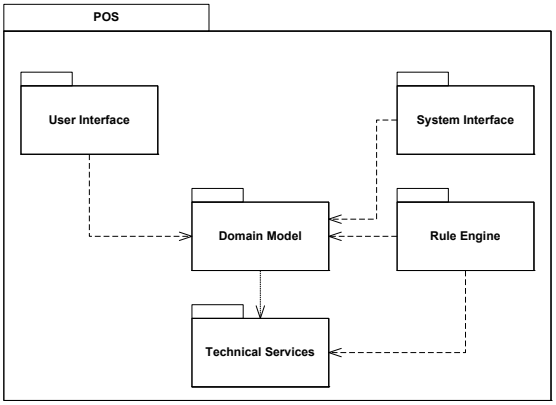

Figure 2: Package overview diagram for the POS system

Dependencies among packages are also shown; these dependencies arise because of relationship among classes in different packages. As an example, consider the association between figure 4 there is an association from classes in Payments to the Customer class of the Sales package. This relationship gives rise to a dependency from the Payments to Sales package as shown in figure 3.

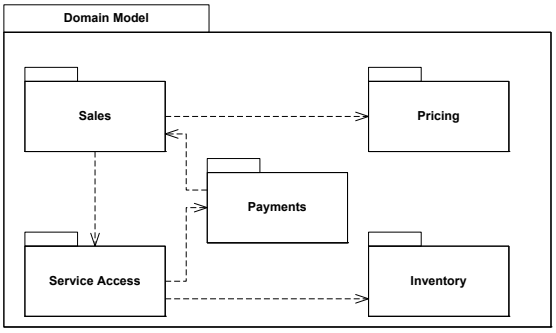

Figure 3: Decomposition of the Domain Model package of the POS system

Typically, class diagrams such as figure 4 will suppress detail and also omit

elements for clarity, since a major purpose of architectural description is communication. In figure 4, e.g., details of methods and attributes of classes have been suppressed and certain classes have been omitted.

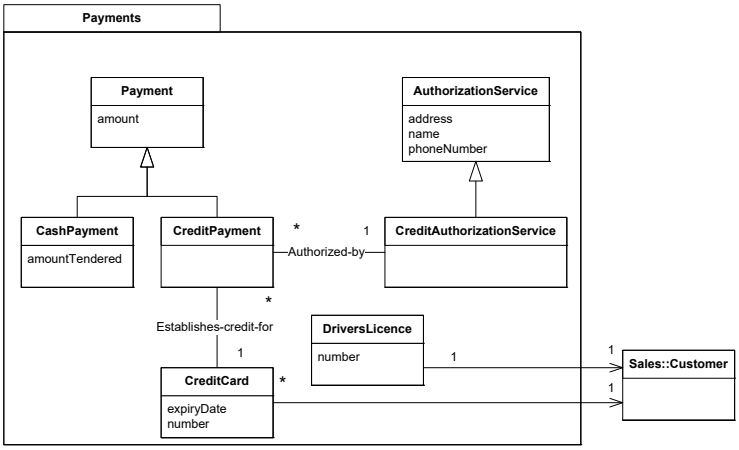

Figure 4: Decomposition of the Payments package of the POS system

3.2 Component and Connectors (C&C) Viewpoint

3.2.1 Concerns

This viewpoint is concerned with the run-time functionality of the system—i.e. what does the system do? This functionality lies as the heart of purpose of the system under development, thus this viewpoint is of course a very central viewpoint, and architectural design often starts from it[4]. In this viewpoint, software systems are perceived as consisting of *components* which are black-box units of functionality and *connectors* which are first-class representations of communication paths between components.

Components embody *functional behaviour* while *control and communication aspects* are defined by the connectors. Paraphrasing this, you can say that components define *what* parts of the system is responsible for doing while connectors define *how* components exchange control and data.

It is important to describe properties of both components and connectors in the documentation. This is done using a combination of textual descriptions (listing responsibilities for example) with diagrams showing protocols, state transitions, threading and concurrency issues as seems relevant to the architecture at hand.

3.2.2 Stakeholder Roles

This viewpoint is important to architects, developers, and may also serve to give an impression of the overall system runtime behaviour to customers and end users.

[4]Hofmeister et al. [Hofmeister et al., 1999] defines a process where this viewpoint is the first to be considered and other viewpoints are derived and elaborated from it.

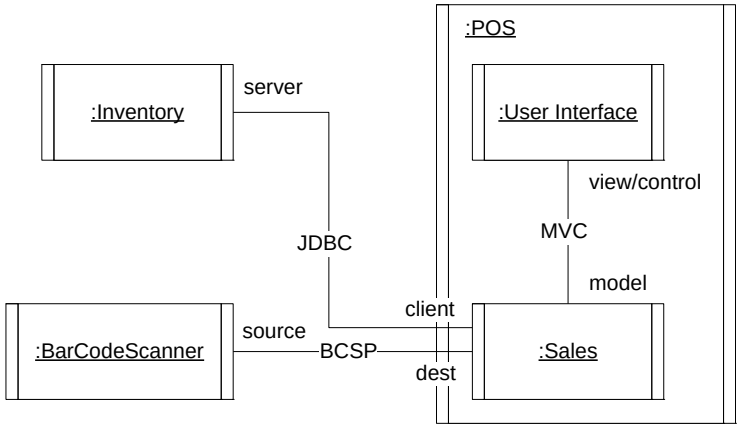

Figure 5: C&C overview of the POS system

3.2.3 Elements and Relations

The C&C viewpoint has one element type and one relation type:

> Component: A functional unit that has a well-defined behavioural responsibility.
>
> Connector: A communication relation between components that defines how control and data is exchanged.

Both are first class citizens of this viewpoint and both may contain behaviour. This is obvious for components, but connectors may exhibit behaviour as well. Examples of connectors with behaviour are those that provide buffering of data between a data producer and consumer, data convertion, adaption of protocols, remote procedure calls, networking, etc.

A connector defines one or more protocols. A protocol defines both incoming and outgoing operations and mandates the ordering of them. Thus a connector's protocol is radically different from a class' interface that only tells what operations its instances provide (not uses) and does not describe any sequencing of method calls.

3.2.4 Example

The POS system has four major functional parts as shown in the C&C view in figure 5. Components are represented by UML active objects, connectors by links with association names and possibly role names. Active objects are typically processes or threads in the operating system or programming language, and links the communication paths between them.

The diagram cannot stand alone, as component names and connector names are only indicative of the functional responsiblities associated with each. We therefore provide an description of component functionality in terms of responsibilities:

- *Barcode Scanner.* Responsible for 1) Control and communication with bar code scanner hardware and 2) notification providing ID of scanned bar code for items passing the scanner.

- *Sales.* Responsible for 1) keeping track of items scanned; their price and quantity; running total of scanned items and 2) initiation and end of sales handling.
- *Presentation.* Responsible for 1) displaying item names, quantity, subtotals and grand total on a terminal 2) printing item, quantity, subtotals and grand total on paper recipt 3) handle key board input for defining quantities when only one of a set of items are scanned.
- *Inventory.* Responsible for 1) keeping track of items in store 2) mapping between bar code ID's and item name and unit price.

Likewise, the connectors' protocols needs to be described in more detail. The level of detail needed depends on the architecture at hand. For some connectors, it may be sufficient with a short textual description (for instance if it is a straightforward application of the observer pattern; or if it is a direct memory read); others may best be explained by UML interaction diagrams; and still others may have a very large set of potential interactions (like a SQL connector) of which only a few may be worthwhile to describe in more detail.

The POS example names three connectors:

- *MVC.* A standard MVC patterns is the protocol for this connector that connects the Sales component serving the role of model and Presentation serving as controller and view.
- *JDBC.* This connector handles standard SQL queries over the JDBC protocol.
- *BSCP.* This connector defines a protocol for connecting with a barcode scanner. Data and control is exchanged using ASCII strings in a coded format containing control words and data elements.

Sequence diagrams can be used to describe connector protocols. Depending on the system, it may be relevant to document connector protocols individually (a sequence diagram for each protocol) and/or to provide the "big picture" showing interaction over a set of connectors. Typical use cases as well as critical failure scenarios may be considered for description.

In our point of sales example, an overall sequence diagram (diagram 6 seems most relevant, as the individual connectors have rather simple protocols. The scenario shown in the diagram is the event of a single item being scanned and registered.

Further detail can be provided, like a sequence diagram showing observer registration and steady state operation for the MVC connector; perhaps table layout or SQL statements for the JDBC; or command language for the BCSP connector. However, most likely this information does not provide architectural insight (they do not affect architectural qualities) and their details should be found in more detailed documentation instead.

3.3 Allocation Viewpoint

3.3.1 Concerns

This architectural viewpoint is concerned with how the software elements of the system – in particular the C&C viewpoint elements and relations – are mapped

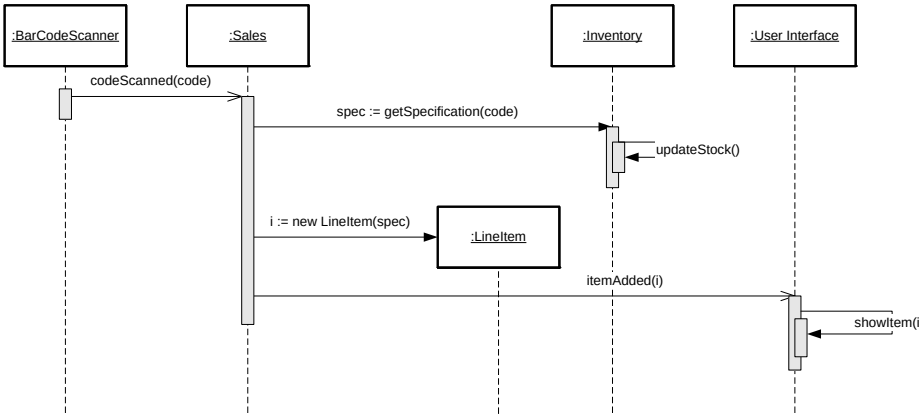

Figure 6: POS "item scanned" scenario

to platform elements in the environment of the system. We are interested in what the software elements require (e.g., processing power, memory availability, network bandwidth) and what the hardware elements provide.

3.3.2 Stakeholder Roles

This viewpoint is important to a number of stakeholders: Maintainers needing to deploy and maintain the system, to users/customers who need to know how functionality is mapped to hardware, to developers who need to implement the system, and to architects.

3.3.3 Elements and Relations

The deployment viewpoint has two primary element types:

> Software elements: These may be, e.g., executables or link libraries containing components from the C&C views.
>
> Environmental elements: Nodes of computing hardware

Furthermore, there are three main relation types:

> Allocated-to relations: Shows to which environmental elements software elements are allocated at runtime. These relations may be either static or dynamic (e.g., if components move between environmental elements).
>
> Dependencies among software elements
>
> Protocol links among environmental elements showing a communication protocol used between nodes.

3.3.4 Examples

Figure 7 shows the deployment of the NextGen POS system using a UML deployment diagram.

The deployment is a typical 3-tier deployment in which presentation is run on a client, domain code is run on a J2EE application server, and data is stored on a database server.

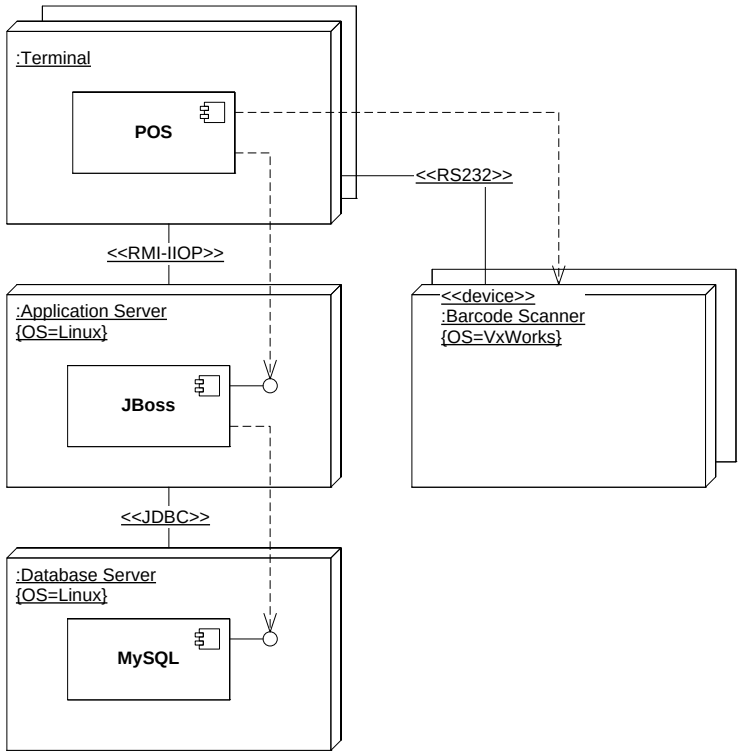

Figure 7: Deployment view of the NextGen POS system

The following elements are of interest

- Environmental elements (shown as UML nodes)
  - The *Barcode Scanner* is the device used for inputting sold items into the system. It is read via an RS232 connection to the POS Terminals
  - The *Terminal* is the main point of interaction for the users of the NextGen POS system
  - The *Application Server* is a machine dedicated for serving all Terminals on an application level
  - A *Database Server* provides secondary storage
- Software elements (Shown as UML components)
  - The *POS* executable component runs the client part of the NextGen POS system including presentation and handling of external devices

(viz., the Barcode Scanner). It communicates with the Application Server via RMI over IIOP

- *JBoss* is an open source application server which is used for running the domain-related functionality of the system. It uses the Database Server via JDBC
- *MySQL* is an open source SQL database which handles database-related functionality (storage, transactions, concurrency control) of the system.

3.4 Overview

The three viewpoints and their associated elements and relations are summerized below.

	Module	CC	Deployment
Elements	Class	Component	Executable
	Interface	-	Computing node
	Package	-	-
Relations	Association	Connector	Allocated-to
	Generalization	-	Dependency
	Realization	-	Protocol link
	Dependency	-	-
+1 view: Architectural requirements			

The mapping to UML is straight forward for the module and deployment viewpoint but less so for the CC viewpoint. For the CC viewpoint, components are show by UML Active Objects (that represent run-time entities with their own thread of execution, typically threads and processes), while connectores are shown by UML links (that represent control- and data flow using some protocol).

Notes

Revision 2.0 by Klaus Marius and Henrik Christensen has updated the notation used to UML 2.0.

Revision 2.1 by Henrik Christensen has updated the notation for active objects.

Revision 2.2 by Henrik Christensen has added the table outlining elements/relations and viewpoints. Idea originally by part-time student Jesper Pedersen at an exam in 2011.

Revision 2.3 by Henrik Christensen has improved appearance of figures (changed format from PNG to PDF).

Revision 2.4 by Henrik Christensen, fixed UML syntax errors in module viewpoint figures. Thanks to Sune Chung Jepsen for pointing them out.

References

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley, 2 edition.

[Clements et al., 2002a] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J., editors (2002a). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley.

[Clements et al., 2002b] Clements, P., Kazman, R., and Klein, M., editors (2002b). *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley.

[Hofmeister et al., 1999] Hofmeister, C., Nord, R., and Soni, D., editors (1999). *Applied Software Architecture*. Addison-Wesley.

[Larman, 2002] Larman, C. (2002). *Applying UML and Patterns*. Prentice Hall, 2 edition.

[OMG, 2003] OMG (2003). Unified Modeling Language specification 1.5. Technical Report formal/2003-03-01, Object Management Group.

[Software Engineering Standards Committee, 2000] Software Engineering Standards Committee (2000). IEEE recommended practice for architectural description of software-intensive systems. Technical Report IEEE Std 1471-2000, IEEE Computer Society.