

Architectural Modeling with the Unified Modeling Language

(extended version, draft)

Harald Störrle*

Ludwig-Maximilians-Universität München
stoerrle@informatik.uni-muenchen.de

1 Introduction

1.1 Motivation

In recent years, the architecture level of systems has been identified as a vital factor in the creation and evolution ("maintenance") of systems and the effective reuse of software artifacts. As Tom DeMarco poignantly put it: "Architecture is a framework for change". At the same time, the Unified Modeling Language (UML) has become *the* standard for expressing designs of software systems. Currently, however, there are very few expressive means for the architecture level in the UML. This paper aims at providing just this. We adopt accepted notations and embed it tightly into the conceptual framework of the UML. The concepts and notations presented here have been used in a practical project with students.

1.2 Issues in architectural modeling

There are several aspects to modeling the architecture of software systems. First, there is an obvious need for clear architectural concepts. Second, an appropriate syntax must be provided to represent them. It needs to be expressive to cover a wide range of systems and architectural styles, yet has to be easy to understand and use. Third, the syntax needs to be underpinned with a precise semantics, so that it is possible to build tools to support working on the architecture level. On the one hand, analysis tools from validation up to formal analysis of an architecture are required. On the other hand,

documentation and cataloging of components, and automated code generation is essential for effective reuse (see [9]). Fourth, the process of developing and evolving an architecture needs methodological guidelines, as a phase in itself and in relation to the overall software development process. Fifth, the quality of the concepts, syntax, semantics, and development guidelines is determined by their practical usability, which is to be assessed by case studies.

1.3 An integrative approach

Obviously, not all of the above issues can be treated exhaustively in this paper. So, we focus on the concepts and syntax. This is a long shot from a true method, or a proper and reasonably sized case-study, however. Also, we omit the semantics completely (see the companion technical report [19] on this issue).

The approach taken here tries to integrate several strands of current work. First, there are some academic approaches: Architecture Description Languages (ADLs) such as WRIGHT [2] feature interesting architectural concepts, but lack convincing graphical representations as well as methodological guidelines. Second, there are commercial approaches such as the ROOM method [16] with a track record of successful practical applications, but without semantical underpinnings and sufficiently clear and advanced concepts. Third, there is the UML [12], subject to enormous interest both commercially and academically, offering a wealth of notions and notations, and emerging methodologies but lacking adequate concepts and syntax for the architecture level.

Our approach is to take accepted concepts, enhance them, and embed them in the UML conceptual framework, so as to make all other

*Thanks go to Alexander Knapp for innumerable inspiring discussions, and to Bernd Gebhard of BMW and Bran Selic of ObjecTime for many helpful suggestions.

UML related work more easily accessible.

1.4 Organization of this paper

Basic concepts for architectural modeling are introduced (Section 2), and a novel notion of component is explained (Section 3). The concepts we have presented are then embedded in the terminological framework of the UML (Section 4). As a concrete syntax for modeling, we introduce *architectural diagrams*, and *functionality*, *realization* and *implementation views* on them (Section 5). First practical experiences with the notions and notations are reported (Section 6). The paper concludes with a summary of the genuine contributions and a discussion of similar approaches (Section 8).

2 Concepts for architectural modeling

The three main concepts are *architectural component*, *architectural interface* and *architectural adapter* (or just: component, interface and adapter, respectively). A *component* is an amalgamation of different *facettes*, that hold information for the activities involving the component (such as retrieval from a store, composition and analysis). As one of its *facettes*, a component holds a set of *interfaces*, which provide a strong encapsulation of the component (i.e. not only the structure, but also the behavior of interfaces, called **role** and **protocol** in ROOM). Components may be accessed only via their interfaces. Connections between components are established by defining *adapters* between their interfaces. An ensemble of components and adapters is called a *configuration*.

2.1 Components, facettes, views

Currently, there is no clearly defined, generally agreed upon notion of component. In our eyes, the characteristic property of a component is *self-containedness*: a component should be self-contained with respect to all activities it might be involved in. This spans all phases of a component's life-cycle. Thus, the activities to be considered include the usual access of a component's functionality in operational use (i.e. any interactions it is involved in), but also e.g. retrieving it from some catalog of off-the-shelf components, and all other software engi-

neering activities relevant for component-based software development (CBSD, see [4] for a brief motivation). Hence, a component is not only a programming-level entity¹, but also an analysis and design entity, and consequently needs to hold all information necessary in these phases, too. Note that this is in accordance with the demand for components as elementary units of functionality relevant to the application domain. The notion of encapsulation and its implications are discussed in greater detail in Section 3. The contents of components is organized in a set of containers called *facettes* (see [14]), that can be aggregated to *views*. While *facettes* are a semantic concept, *views* are a syntactic abstraction, introduced to support specific usages.

2.2 Interfaces

An important contribution to the self-containment of components is made by its *interfaces*. Any interactions a component engages in are carried out exclusively via its interfaces. Interfaces can be distinguished by the kind of connection they allow. On the one hand, there are connections between peers, as pointcast or multicast connections. On the other hand, connections can be used for broadcast-type connections. Therefore, we distinguish between *ports* for the former type of connections, where all communication partners are known, and *service access points (SAPs)*, for the latter, where communication partners may be anonymous.

2.3 Adapters

Adapters connect interfaces in a completely passive way. They are assumed to be of perfect quality (i.e. exhibit no faulty behavior). Adapters specify the sets of signals they may transmit, as well as the behavior of the interfaces that are connected by adapters. Adapters connect at least two interfaces.

2.4 Configurations

An architectural configuration is an abstract representation of a system's structure. It con-

¹Of course, a component should also not only hide its implementation, but indeed its implementation technology, i.e. components implemented in non-object-oriented languages are perfectly acceptable. Relying on any specific OO-concepts (such as inheritance as specialization) is a breach of the secrecy principle.

sists of a set of architectural components, that are loosely coupled by adapters. Different styles of configurations can be achieved. Apart from arbitrary networks, layered systems are also frequently encountered.

3 Architectural components

3.1 Requirements

The most important feature of an architectural component is self-containment. This means, that an architectural component is a unit, that provides a strong, explicit and rich encapsulation, leading to three major requirements.

As for strength, access to a component must be limited to specific interfaces. As a first requirement, architectural interfaces must specify not just a signature, but also a behavior (commonly termed 'protocol'). It can be seen as a specification of guaranteed behavior of some interface. So, changes to the internal features of a component that affect its behavior can be localized before they invalidate any usage-relationships the component is already involved in. Self-containedness, in other words, is essential for the compositionality of components.

As for explicitness, the interface must exert access control in both directions, i.e. it is not enough to restrict access *to* a component, but also, the access *from* a component must be controlled. This latter requirement is vital with a view to reusing components: in order to factor out components from an existing application, it must be disentangled from the web of interrelated components. This is can be very difficult, if the dependencies are not clearly defined. In order to achieve true self-containedness, it is much better to make explicit the components (or rather: interfaces) a component depends on than to just refer to them in the code, say, whenever needed.

For the third requirement, the richness of the encapsulation comes into view. Self-containedness of components should apply to all software engineering activities the component is involved in, throughout all of its life-cycle and throughout all of a system development life-cycle. Obviously, the creation of an architecture occurs early in system development, and so this includes the analysis-phase representation of a component as well as (versions of) its implementation to any information needed for

maintenance and reuse. At least the following activities must be supported by a component:

- **Retrieval:** For a specific site in an architecture of some project, suitable components must be retrieved from (large) databases of off-the-shelf components. This requires specifying a profile of the candidate component
- **Match Quality Assessment:** Once retrieved, it is necessary to assess how well a component matches the specific site under consideration. While retrieving components with a given profile ensures certain coarse matching criteria (e.g. right hardware, right keywords wrt. functionality and so on), the matching of interfaces has to be investigated with greater scrutiny (and greater computational effort).
- **Tailoring:** Typically, the selection will yield a non-perfect match, its quality being revealed by the previous assessment. Supposing that the selected component is to be used anyway, it needs to be adapted. There are basically three complementary types of adaptations to be considered. First, components may have switches and parameters that could be used for customization. Second, there are simple adaptations like aliasing of signals, transposition and projection of parameters, or simple pre- or post-processing that can be done by a wrapper. Third, a component may need real modifications, like a restructuring. This is best accommodated as a specialization.
- **Design Debugging:** Next, the choice and adaptation of component has to be evaluated. While a successful match quality assessment and tailoring ensure that interfaces match locally, the global interplay of components needs to be assessed as well. For instance, often a load balancing analysis is required.
- **Cataloging:** Finally, components need to be stored for retrieval, i.e. according to some cataloging scheme. It is highly desirable, that there be automated cataloging procedures.

To accommodate these activities, architectural components need to incorporate much more information than is present in today's components

(like JavaBeans, COM or CORBA components). Also, this information must be represented semantically, so that automated tools will be able to make use of these informations. At the same time, there must be abstractions of components to support the activities of human users. These two requirements can be realized by structuring components as a set of facettes, that store the information for the activities listed above, and aggregating them into views as needed.

3.2 Facettes

Facettes can be used as a means for classifying items, specifically those that are subject to reuse [14]). In the setting of CBSD, reuse is of course the primeval purpose, so that it appears to be a rather adequate name for the partitions of components.

- **Functionality:** The functionality facette captures the purpose(s) of a component, its system boundaries, and the dependencies to adjacent systems.
- **Interfaces:** The interfaces facette contains a set of interfaces, i.e. contracts about the component's behavior. It corresponds to a port in the ROOM terminology. An Interface consists of a signature (i.e. the sets of ingoing and outgoing signals) and a behavior specification.
- **Realization:** A realization facette could be either a structural refinement or a behavior specification. This facette is very much like the capsules/connectors of UML/RT, the actors/connectors of ROOM, or the component/connectors of WRIGHT.
- **Platforms:** This facette holds the requirements of the execution platform, i.e. hardware, operating system, a particular process- or scheduling model and additional required software like libraries.
- **Implementations:** This facette contains a family of executables for various platforms and environments, and possibly also subsequent revisions of executables for downward compatibility.
- **Parameters:** Many components need to be/can be parameterized for different contexts. Also, many components need calibration. The parameters facette holds data of this kind.

- **Quality of Service:** This facette specifies required qualities of underlying services, such as network bandwidth, memory capacity etc.

Components may be generalized to component types, templates or prototypes, that can be instantiated or cloned. In a sense, components correspond to objects as in object-oriented programming languages (and the usual designations including the UML). However, there are important differences.² First, objects are very fine grained entities, mostly of a technical nature. Components, on the other hand, must be able to represent very large real-world entities. Second, objects clearly violate the three requirements we have put up. Thus, objects are an inappropriate concept for architectural modeling. An algebraic specification does better in that it specifies the behavior of an interface by a set of axioms³, and offers an abstract representation for the component (cf. [21]). It still violates the second, though, and is very hard to use for the vast majority of designers. Also, it focuses exclusively on the transformational aspect of a component, and leaves out any information for other phases in a component's life-cycle.

3.3 Views

Some simple examples of views and their application are the following.

- a *functional view* documents a system's boundaries and external functionality. It can be used for a top-level architectural presentation of a system, as it is necessary for designing an architecture, but also for a classical functional decomposition in the style of structured methods.
- an *architectural view* may present the system boundaries, its external interfaces, the substructure of the system's components and the specification of behavior for basic components without further structural refinement. This view can be used to support the transition from a functional to a structural view of a system. Separating independent subsystems, it is also a valuable help for concurrent engineering.

²For a discussion of the differences between objects and components in the sense of e.g. Java Beans, see [13].

³Recently, there have also been advances in specifying the behavior of algebraic specifications in a more finely grained way, e.g. by co-algebraic approaches.

- an *implementation view* may present the executable code that implements a components architecture together with dependencies among code units. This view may be used to determine compile- and load dependencies, i.e. it can be used to generate `make`-files or similar structures.

All three of these views may be either flat, i.e. representing only one layer of functionality, realization or implementation, respectively, or the recursive sub-functionality, -realization or -implementation, respectively, to arbitrary depths.

Note that the implementation view corresponds to the information conveyed by components in the sense of Java Beans, COM, CORBA, or UML components. The realization view corresponds to a model similar to those of UML/RT and ROOM.

Other applications and activities might require further views and facettes. For instance, there might also be a deployment view, a hardware/software distribution view, a samples view (e.g. pointers to sample applications with the component, experience reports), a performance view (e.g. load figures relative to number of users and so on), a test view (e.g. test cases, partitions of input data, error density and so on).

4 Mapping of concepts to the UML Abstract Syntax

In this section, the concepts introduced above are embedded into the conceptual framework of the UML. This is done by introducing new classes of the UML's meta-model as stereotypes of existing classes. This mechanism is the preferred way of extending the UML, according to the UML semantics definition (compare the chapter on extension mechanisms, [12, p. 2-65]). Figure 6 shows an excerpt of the UML meta-model, and the concepts we have introduced before.

Protocol Protocols have a fixed set of at least two roles, a collaboration between these, and possibly some matching interactions. A role is an UML actor with a UML state machine and two UML interfaces, tagged as ingoing and outgoing, respectively, to discriminate the two kinds of dependencies that are documented in interfaces.

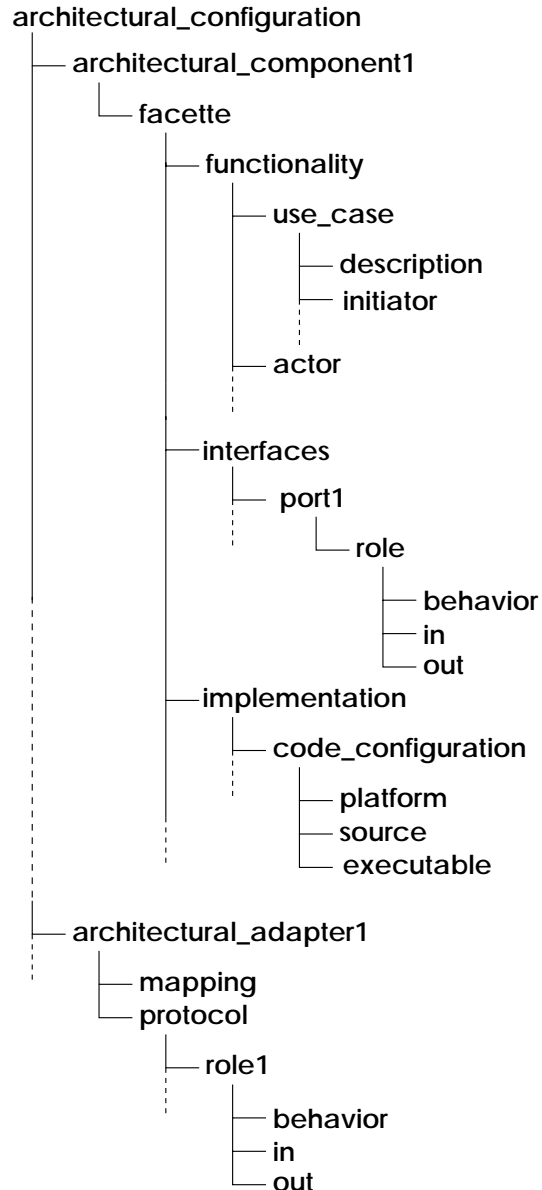


Figure 1: A sample abstract syntax representation of a configuration. All the views in Figure 3 can be generated from this tree.

Architectural interface An Architectural interface has a role it plays in whatever connections it is used in.

Role A role is an UML package, containing two UML interfaces, tagged as incoming or outgoing, and a behavior specification, i.e. an UML state machine.

Architectural facette An Architectural facette is an UML package with a tagged value called *FacetteKind* that can take on any of the values *Functionality*, *Interfaces*, *Realization*, *Platforms*, *Implementations*, *Parameters* or *QoS*. Facettes with different FacetteKinds can be seen as subclasses of the class Architectural-Facette. Each facette has individual restrictions depending on the value of FacetteKind:

- The *functionality facette* contains a set of UML actors and use cases. Actors are understood in the original sense of OOSE- or ROOM-actors, i.e. they are arbitrary external systems, including human users and other software and hardware components.
- The *interfaces facette* is a collection of architectural interfaces.
- The *realization facette* roughly corresponds to the realisation elements of a UML subsystem. It contains either an architectural design or a behavioural specification, the latter being a plain UML state machine that uses only the operations of the interfaces in the interfaces facette. The former consists of an architectural configuration that is connected to the interfaces of the component by links specified in the adapters facette. A realisation facette also contains a set of extension and customisation points, i.e. a description of the parameters and switches that may be used to adapt the component to a particular application site.
- The *platforms facette* indicates the required platform a component needs to be executed, i.e. a certain type of hardware, operating system etc. In order to structure this facette, the tagged values *Hardware*, *OperatingSystem*, *Compiler*, *ComputingPower*, *Memory* and *StorageCapacity* are predefined. Further aspects may be added as needed.
- The *implementations facette* contains a set of binaries (possibly also sources) for a component, tagged with information like the target platform, version numbers etc. Every implementation may consist of a set of UML components, possibly again grouped in UML packages, and any uses- imports or refinement-relations between them.

- The *parameters facette* can be realised by an application specific set of tagged values.
- The *quality-of-service facette* specifies any guarantees a component makes with respect to the quality of the services it provides, e.g. the response times for certain pairs of incoming/outgoing signals. Currently, the UML offers little expressive means to model qualities of service. For the time being, it could be represented by a set of tagged values.

Architectural views Architectural views are UML packages.

Architectural components The concept of architectural components is a refinement and extension of the UML notion of subsystem. UML subsystems offer UML interfaces and operations (more generally: UML features), its contents is divided into specification and realisation elements, both of which are represented by arbitrary ModelElements. Architectural components, in contrast, offer architectural interfaces only. Their contents is organised into a number of well-separated facettes with distinct purposes and specific constraints. Architectural components have the additional boolean attribute *is-Static*, that is false, if any interface has a multiplicity other than the fixed value 1. Components also have the boolean attribute *hasServices*, that is false, if they have no broadcast interfaces. Static components without broadcast interfaces can be used as layers in a layered system.

Architectural adapter Architectural adapters are connections between architectural interfaces. They correspond directly to UML associations, but also have a protocol, whose roles correlate to UML association roles.

Architectural configuration An Architectural configuration consists of a collaboration of components. Layered systems can be modelled by an acyclic graph of static components with broadcast interfaces between them.

5 Architectural diagrams

In this section we present a concrete syntax for the concepts presented above. These diagrams

are called architectural diagrams. Since all concepts presented here are mapped to the UML, the corresponding usual graphical representation in the UML should be used where possible. So, all concepts presented here have an explicit representation which follows the conventions of the UML concepts they are mapped to. However, most concepts also need different representations for different pragmatic situations, i.e. the representation depends on the usage context, i.e. on the view a concept appears in. Here, we try to stick to other accepted notations, such as those of the ROOM method (see [16], the notation used there was originally presented in [5]) and UML/RT (see [8] for a concise description of the syntax). In some places, however, slight proprietary modifications are unavoidable.

Interfaces As an explicit representation, the usual UML packages can be used; the signal sets are simply UML interfaces tagged as in and out, respectively. In the context of a realisation view, the UML/RT conventions are adopted (see Figure 3 (c,d), where pointcast interfaces appear as little boxes on the edges of bigger boxes). In the context of an implementation view, interfaces should be represented in the lolly-notation (see Figure 3 (f)). Broadcast interfaces have no direct equivalent, neither in UML nor in ROOM, so we made up the symbol \square for this purpose. Dynamic structure is represented by stacked interfaces symbols as known from ROOM.

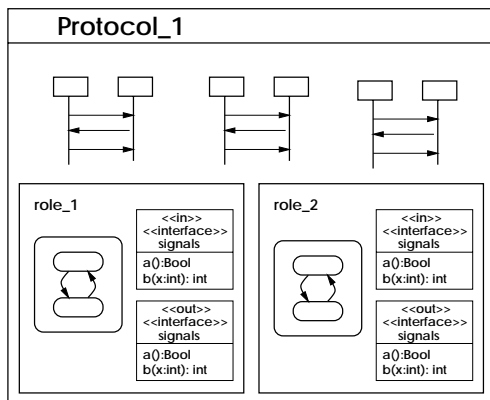


Figure 2: Definition of a protocol.

Components, facettes and views Components are represented by boxes with a small compartment holding its name, much like UML

classes. The other representational features are determined by the view one has on a specific component.

- The functionality view can be adequately represented by the usual use case diagram. Associated documents with natural language descriptions of roles, use cases, and interfaces can be represented by document icons associated to model elements of the use case diagrams.
- The architecture view is represented in the syntax known from ROOM and UML/RT (see [8] for a concise description). Often, there is a dedicated controller sub-component for a component. Semantically, this can be represented by a subcomponent with a broadcast port to its peers. As some syntactical sugar, the notation known from ROOM can be adopted.
- The implementation view is represented by the usual UML syntax, i.e. by package and component diagrams.

All three views can show a flat component or a number of subsequent levels of refinement. See Figure 3 for some examples of flat and deep representations of functionality, architecture and implementation views. In general it is possible to mix views, both for peer and for sub-components.

Adapters Adapters are represented as plan lines between the components they connect. Frequently, adapters need to perform simple actions such as renaming, transposing or projecting from the signals transmitted by them. In the conceptual body presented here, this can be done only by an intermediate component. Such a component, however, has a trivial structure, so that often it is convenient to use syntactic sugar instead. Similarly, sometime one needs to model non-perfect connections. Again, intermediate components clutter up architectural diagrams and should be replaced by an alternative notation. See Figure 4 for some suggestions.

Configurations Configurations are represented as collaborations of components, where the representation of components depends on the view chosen on the respective component. Layered systems can be represented by an alternative syntax (see Figure 5).

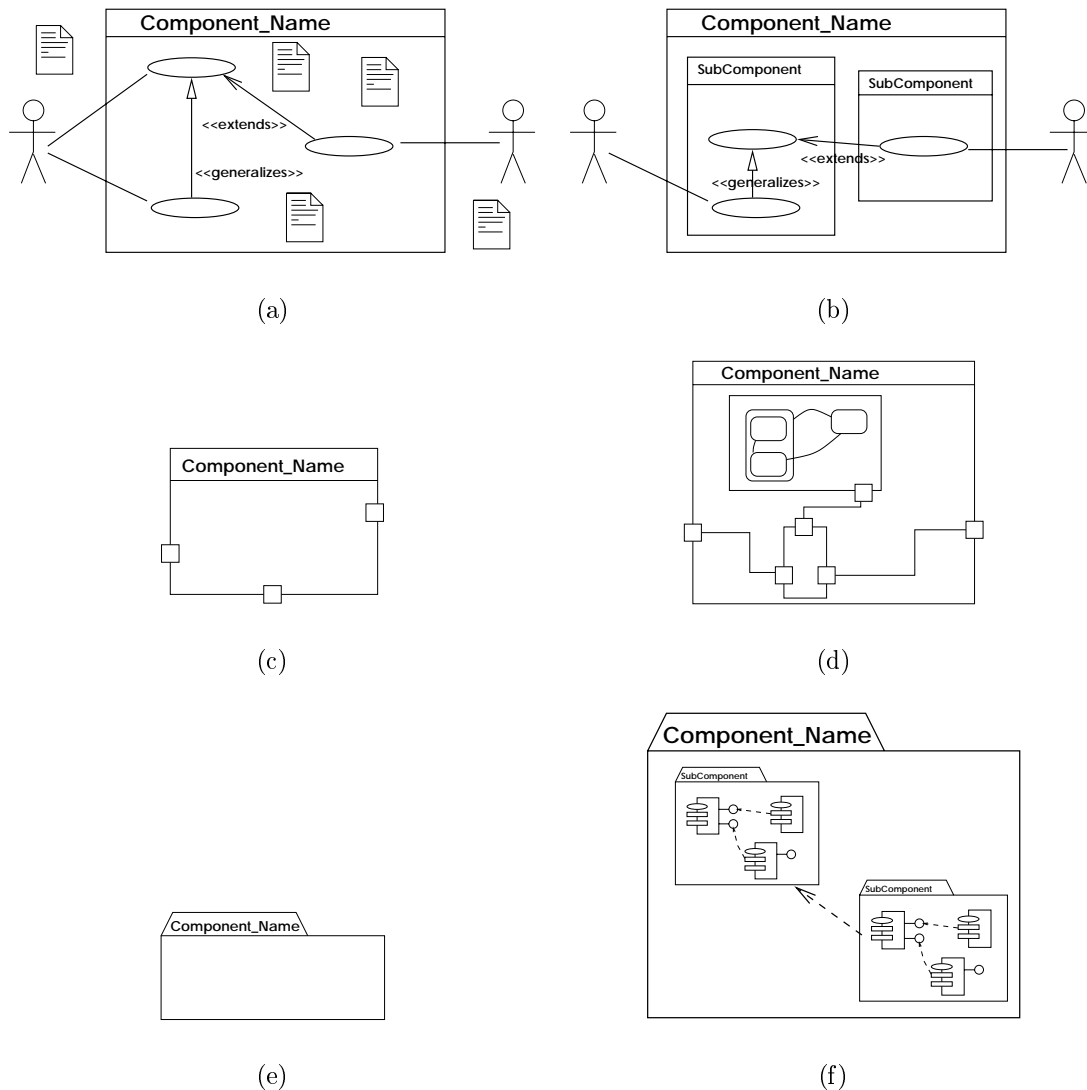


Figure 3: Some views on a component: flat and deep representations (left to right); functionality, realisation and implementation views (top to bottom).

6 Practical experiences

The concepts and notations introduced above have been used in conducting a one-term practical assignment with undergraduate students. Architecture diagrams have been used for the initial definition and subsequent evolution of the system’s architecture. So far, they have been used only on a paper-and-pencil basis, i.e. without editors or analysis tools.

Still, all participants have found architecture diagrams helpful and indeed very adequate for documentation of and communication about the

overall system. The concepts were generally acclaimed as being readily understandable and useful, even in the absence of automated tools. Unsurprisingly, the syntax revealed some deficiencies, in particular wrt. to timing annotations and dynamic structure. It proved to be necessary to guide the students very tightly through the development process, though the incidental occurrences of “diagram abuse” largely resulted from UML diagrams rather than from architectural diagrams.

The most valuable effect stems from having different views of the *same* architectural enti-

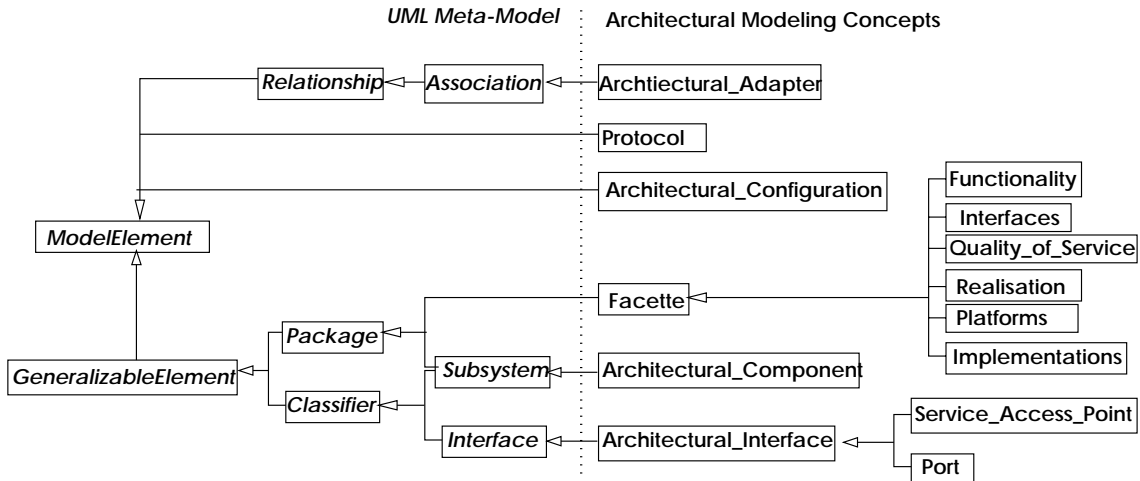


Figure 6: Generalisation hierarchy of the main concepts for architectural modelling (in UML notation). UML-Metaclasses are in slanted type.

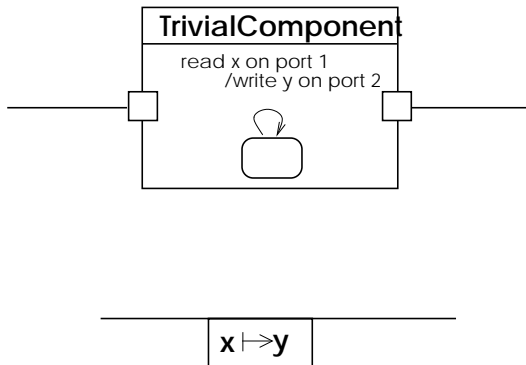


Figure 4: Alternative notation for connections with trivial behaviour.

ties during the complete development life-cycle. It was particularly helpful, that these entities were described not only with the usual static interfaces, but also with specifications of their behavior (i.e. the interfaces' protocols). This greatly improved the separation of concerns and turned out to be effective support for concurrent engineering, and thus was pivotal for the overall system development process. For instance, with behavioural specifications of the interfaces, it is possible to stage interactive role plays and walkthroughs on the architecture level. While already present in many approaches to architectural modeling (e.g. [16, 2, 1, 7]), behavioural specification of interfaces have their greatest impact when used very early, as it is encouraged

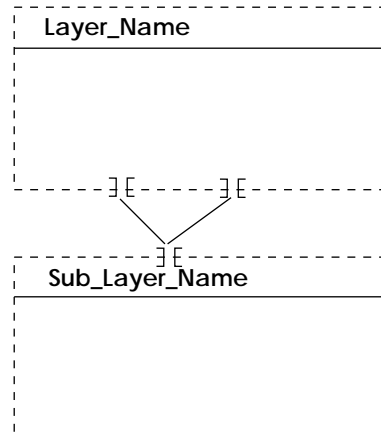


Figure 5: Alternative notation for layered systems.

by the tight integration of functionality and realisation views provided by this approach.

7 Other approaches

There are a number of textual Architecture Description Languages (ADLs), most notably WRIGHT [2], UNICON [20], Rapide [7] and SARA [5] (see [10] for an overview). Some of them also have ad-hoc graphical notations and associated tools for code generation or analysis. None of them relates to the UML, however, or is integrated with a broader notational

and methodological framework as is present and emerging with the UML, respectively. The same applies for other formalisms, such as SDL and HOOD [15].

The UML itself offers only very scarce facilities for modelling on the architecture level. The present version (1.1) has only the implementation diagrams (package, component and deployment diagrams) and traditional Use cases. The forthcoming version (at the time of writing, the most recent revision is 1.3 beta 7) also offers subsystems, but their semantics and integration with the rest of the UML is largely unclear. A real-time profile for the UML has been discussed, adopting notions from the ROOM method (see [17, 8]), though no publication on this issues is currently available.

The novel notion of component presented in this paper⁴, on the other hand, features strong encapsulation, direct support for the activities relevant in CBSD, and an integration with accepted design notations. There are quite a number of publications on the definition of software engineering processes for CBSD (see e.g. [3, 6, 11]). Surprisingly, however, none of them discusses the requirements on the underlying notion of component generated by the activities used in the process, let alone introduce a similar notion of component as is presented here.

8 Summary

This paper provides an embedding of practically proven concepts into the conceptual framework of the UML. It also enhances the notion of component so as to better accomodate the activities specific to CBSD. The concepts and notations have been used in a practical project with students, though so far, insufficient data has been gathered.

Therefore, future work will concetrate on implementing a tool to support the diagrams and the CBSD-related operations. Also, furhter work is required on the semantical foundations of architectural modelling (see the companion technical report [19] for a semantics of ROOM-diagrams).

⁴A previous version of the component notion has been presented in [18].

References

- [1] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.
- [2] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM TOSEM*, 6(3):213–249, 1997.
- [3] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A componentware development methodology based on process patterns. In ??, editor, *Proceedings of the 5th Annual Conference on the Pattern Languages of Programs, (PLOP)*, 1998.
- [4] Francois Bronsard, Douglas Bryan, W. Voytek Kozaczynski, Edy S. Liongosari, Jim Q. Ning, Asgeir Olafsson, and John W Wetterstrand. Toward software plug-andplay. In Mehdi Harandi, editor, *Proc. 1997 Symp. on Software Reusability (SSR'97)*, 1997.
- [5] G. Estrin, R.S. Fenchell, and M.K. Razouk, R.R. Vernon. SARA (System Architects Apprentice): Modelling, Analysis and Simulation Support for Design of Concurrent Systems. *IEEE Trans. Softw. Eng.*, 12(2):293–311, 1986.
- [6] Wolfgang Hesse. From WOON to EOS: New development methods require a new software process model. In A. Smolyaninov and A. Shestialtynow, editors, *Proc. WOON'96/WOON'97, 1. and 2. International Conference on OO Technology*, pages 88–101, 1997.
- [7] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *IEEE Transactions on Software Engineering*, 1980.
- [8] D.C. Luckham, L.M. Augustin, J.J. Kenney, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture using Rapide. *IEEE Trans. Softw. Eng.*, 21(4):336–355, April 1995.
- [9] Andrew Lyons. UML for Real-Time (Overview). Technical report, ObjecTime INC., see www.objecttime.com, 1998.

- [10] Masao Matsumoto, Satoshi Hayano, Takahiro Kudo, Hideo Yoshida, Shunzo Imai, and Kiyoshi Ohshima. Specifications Reuse Process Modelling and Case Study-Based Evaluations. In *Proc. 15th Annual Intl. Computer Software and Applications Conf. (COMPSAC'91)*, 1991.
- [11] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. Number 1301 in LNCS. Springer, 1997. also appeared in SE Notes, Vol. 22, No. 6, pages 60–75.
- [12] Jim Q. Ning. A Component-Based Software Development Model. In *Proc. 20th Annual Intl. Computer Software and Applications Conf. (COMPSAC'96)*, 1996.
- [13] OMG. OMG Unified Modeling Language Specification (draft, version 1.3beta R1). Technical report, Object Management Group, 1998. available at uml.shl.com.
- [14] Cuno Pfister and Clemens Szyperski. Why Objects Are Not Enough. In Hans Blubber, editor, *Proceedings First Intl. Component User Conference (CUC'96)*, July 1996.
- [15] Ruben Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5), may 1991.
- [16] Peter J. Robinson. *HOOD – Hierarchical Object-Oriented Design*. Prentice Hall, 1992.
- [17] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real Time Object Oriented Modeling*. Wiley & Sons, Inc., 1994.
- [18] Bran Selic and James Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, Rational Software Corp. & ObjecTime Ltd., 1998.
- [19] Harald Störrle. A different notion of components. In Andreas Schürr and Peter Hofmann, editors, *Object-Oriented Modelling of Real-Time Systems (OMER'99)*, 1999.
- [20] Harald Störrle. A formal semantics of ROOM with applications. forthcoming, 1999.
- [21] Jan van Leeuwen, editor. *Formulations and Formalisms in Software Architecture*, volume 1000 of LNCS, pages 307–323. Springer, 1995.
- [22] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Handbook of theoretical Computer Science*, pages 675–788. Elsevier, 1990.

A Some guidelines for using architectural components and associated concepts

The modeling concepts presented in this paper have been tested in a one-term practical assignment of a group of students. Their task was to create a client/server system using a number of predefined components. They had no computer support for modelling (let alone analysis) of architectural designs, and had only a vague idea of how the diagrams were supposed to be used. The most important lessons learned from the project were:

- Architectural modelling must occur early in the systems development, and it must be perpetually improved and maintained.⁵
- Architectural designs are an important means for communication among development teams, and between development teams and the customers (in this case, the supervisors). The structural decomposition of components can be used to assign tasks to people (and processors), and to improve separation of concerns.
- We also learned that the benefit of architectures lie in its simplicity, and that it is well worth drawing the simplest of diagrams, as long as there is a consensus on what it means. Even without a clear meaning, diagram aesthetics, surprisingly, turned out to be a good means of assessing the quality of an architectural design.

Based on the experiences of the practical reported above, we propose some guidelines for using architectural designs, and the views defined above, in particular. These fragments can be seen as a set of process patterns⁶ These guidelines have not yet been tested seriously. Obviously, they require further study and practical tests, so this is to be read as a first proposal only.

⁵This is not only intuitive, but a confirmed empirical fact gained in large reuse-projects: "Reusability at this [the design] level is more advantageous than at program language level [...]" (see [9, p. 502]).

⁶A different, more elaborate approach to process patterns for component-based software development is presented in [3].

A.1 Component Analysis

In the context of component-based software development, determining the structure of a component, i.e. the component analysis, is the most important step. Also, it is the first activity, that is used to trigger all other activities. Therefore, we treat this activity more extensively than the others, and also use it to demonstrate the usage of the different views introduced above.

A component analysis is subdivided into the following five steps:

- First, the system boundaries need to be established in a **context analysis**. The main result are the actors interacting with the system, that can be documented in a context diagram, see Figure 7.
- Then, the use cases of the systems have to be determined **functional analysis**. This is typically, where component catalogs can be consulted the first time to find out whether predefined components for some functionality exist. Functional analysis can be guided by checklists for the descriptions of roles and use cases. First, all roles need to have precise descriptions of their responsibilities. This can be supported by real-life scenarios expressed in natural language. Then, for each use case, the initiator role, non-functional requirements, abstract parameters, abstract pre- and post-conditions and exceptions have to be determined.
- Next, the relations between roles and use cases need to be determined by a **relational analysis**. The UML offers uses-, extends- and generalizes-relationships for use cases, and generalizes-relationships for roles, though other relationships might be useful as well. Functional and relational analysis result in a fully-blown use-case diagram, see Figure 8. The relational analysis exhibits clusters of related use cases.
- These clusters can be used in a **structural analysis** to structurally decompose a system. Any separatable cluster should be turned into a architectural component, possibly aggregating some components to a more global unit, or immediately refining and re-allocating some components. As a rule of thumb, if there is a generalization between two use cases, they can be realised by delegation within a single component.

Also, deployment can be used as a guide for structural decomposition. Now, a component should contain a small number of use cases connected with generalisation-extends-relationships, see Figure ??.

If the component subject to structural analysis is not to be refined any further, instead of a substructure a specification is provided as a state machine, and the next step is skipped.

- If a subconfiguration of a capsule has been arrived at, the interplay of the subcomponents must be elicited by a **collaboration analysis**, i.e. turning the configuration that resulted from structural analysis (ignoring the use cases, i.e. as in Figure ??) into a collaboration diagram (i.e. supplying interactions for a collaboration, both in the sense of the UML metamodel).

After the structural and collaboration analysis have been completed, a component abstraction may be initiated.

A.2 Match Quality Analysis

When a looking for a component to fill some gap in a design, possible candidates must be analysed for the degree of fitness.⁷ Given the concept of architectural interface, the match quality can be analysis one interface at a time. Since the number of candidates should be expected to be enormous, it is vital to reject non-matching candidates as quickly and cheaply as possible. Therefore, first the signatures should be checked, before the behaviour is analysed. Again, there are several techniques that come into question – trace equivalences, bisimilarity checks, and partial order techniques, depending on the application setting.

A.3 Change Impact Analysis

When one component is to be replaced by another (possibly just a new version of the same component), the impact of this change needs to be assessed. Covering the complete behavior of a component (i.e. the composition of all interfaces and the internal structure) rather than just individual interfaces. Change impact analysis is

⁷This seems to be similar to to the binding strength of biomolecules at docking sites.

much more expensive than match quality analysis, so that should be used sparingly, i.e. only after a candidates has passed the match quality analysis successfully.

A.4 Component abstraction

The result of a component analysis is a glass box component, i.e. a component with all necessary analysis results present, but scattered over the component. Component abstraction aggregates the dependencies of this component, and the dependencies that any users of this component will have. It results in the architectural interfaces being created. The interfaces ensure, that the component is indeed self-contained, an essential for compositional analysis, and for smooth collaboration between different development teams. See Figure 12 for the result of joining the data from collaboration analysis and structural analysis, the last two steps of the component analysis process pattern.

After a component abstraction, the component can be allocated to some team to work on it independently of all other groups. In their work, they apply again the same process from steps from analysis through integration, only on a smaller scale. The overall development process has been termed recursive (see [?]) or fractal (see [6]).

A.5 Component realisation

Once the analysis and abstraction of a component are completed, one has a rather precise specification, which can be used for the realisation of architectural components. Realisation can mean creation of code or procurement, i.e. purchase, extraction from some other system, or outsourcing. In the case of creation of code, all standard techniques and procedures apply. Depending on the abstraction level and the performance requirements for the component, at some point, at least part of the component realisation should be automated.

A.6 Component integration

Given that all subcomponents of a configuration have been either successfully realised or – in case of predefined components – procured, the integration can start. Visually, integration is the replacement of actors by another subsystem, see

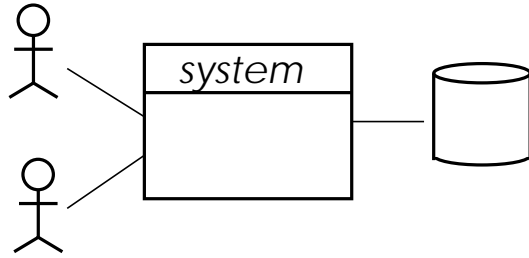


Figure 7:

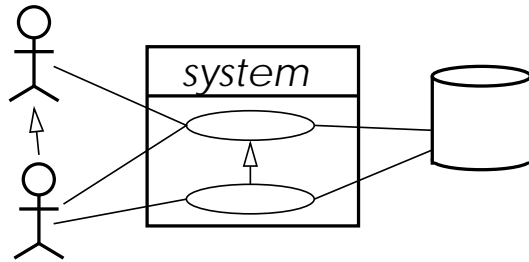


Figure 8:

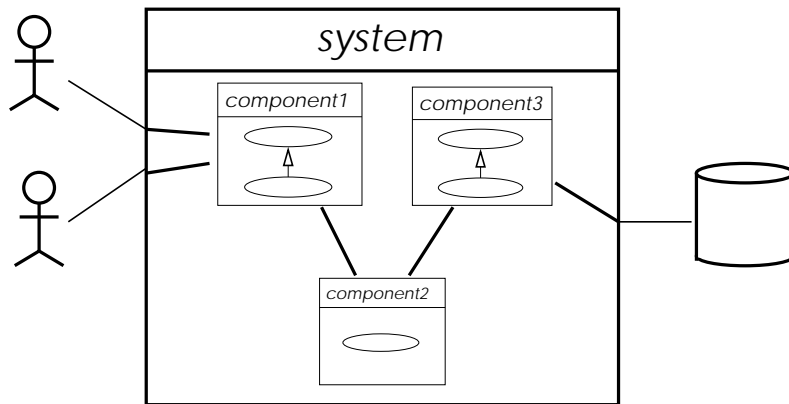


Figure 9:

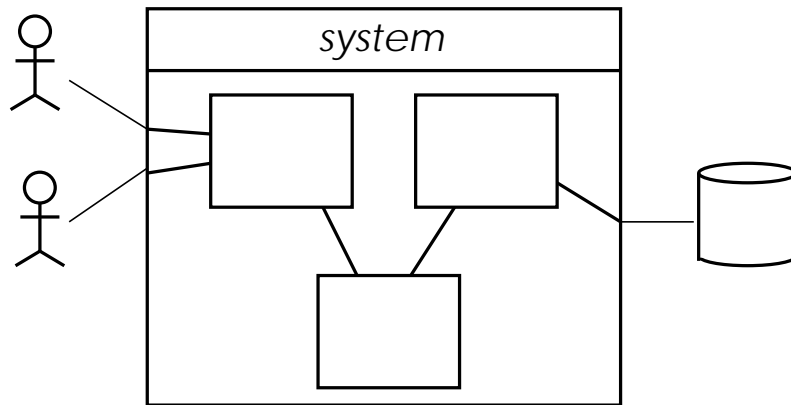


Figure 10:

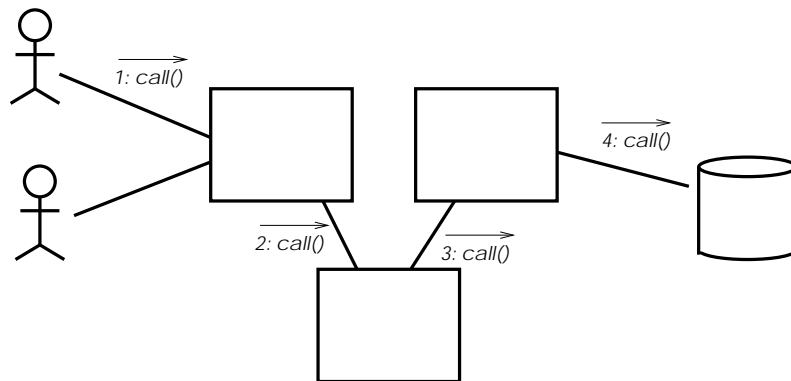


Figure 11:

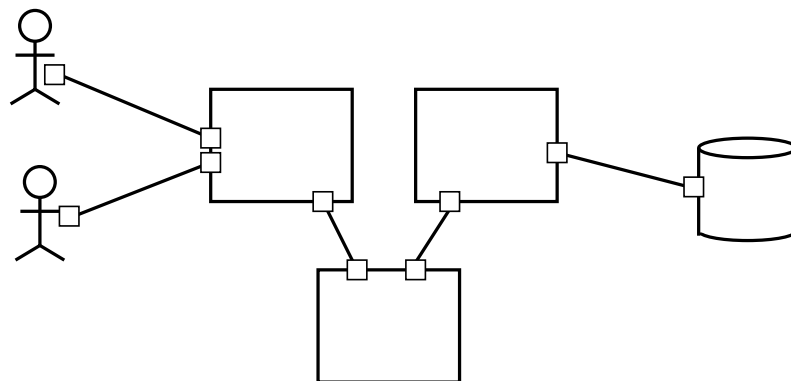


Figure 12:

figure 13.⁸ The integration process gives rise to the deep views (see Figures 3 (b,d,f)).

The functionality view can also provide good starting points for walkthroughs. Its interfaces view can provide the data for automatic test case generation. The component integration process should be carried out in two steps, as pointed out above, with match quality analysis preceding more complete checks, i.e. change impact analysis.

Sometimes, analysis during integration might suggest to shift subcomponents across the boundaries of their enclosing components, either to capture the adequately capture the logical structure of the system, or to satisfy technical requirements (such as minimising network traffic). Then, one has to go back to the respective structural decomposition step before continuing.

⁸Note, that the actors of ROOM were renamed to capsules in UML/RT due to the name clash with UML actors. UML subsystems, on the other hand, are a kind of classifier, i.e. they share most properties with both UML actors, and UML classes (that are used in [8] to model capsules). Architectural components can play the role of capsules (in the realisation view), actors (i.e. bounding systems of a component), and really are subsystems.

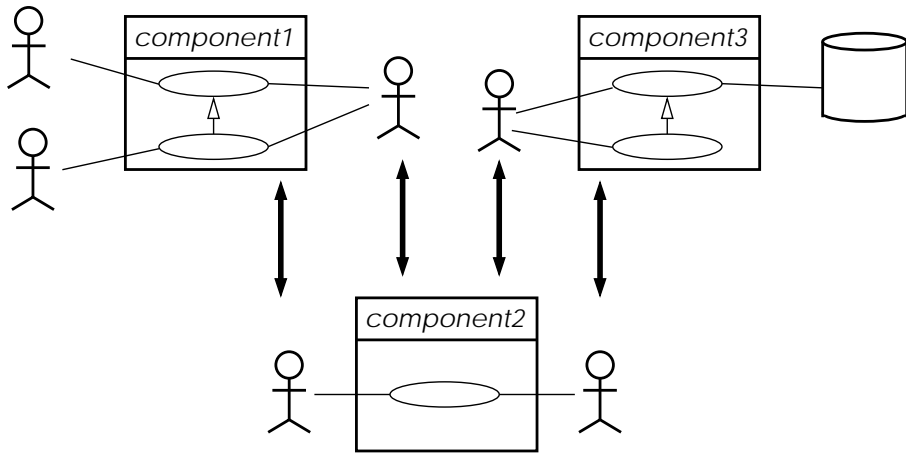


Figure 13: Visualisation of integrating subcomponents into a configuration.