Available online at www.sciencedirect.com
SCIENCE DIRECT®

ELSEVIER

Information and Software Technology 48 (2006) 456–470

INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

A framework for classifying and developing extensible architectural views

Jungwoo Ryoo[a], Hossein Saiedian[b,*]

[a]Information Sciences and Technology, Penn State University, Altoona, PA 16601, USA
[b]Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS 66045, USA

Received 16 December 2004; revised 12 April 2005; accepted 11 May 2005
Available online 16 August 2005

Abstract

Despite its widespread use in the software architecture community, architectural views and relationships among them are poorly defined. A solid taxonomy of views is a critical factor in tackling this problem since it must adopt an unambiguous definition of views and provide rigorous criteria for classification. Nevertheless, the existing taxonomies of views fail to eliminate vagueness surrounding the definitions of views and their inter-relationships mainly due to their informal nature. One of the most significant consequences of these failures is inability to systematically define new views in support of domain-specialization. This paper is an attempt to resolve these outstanding problems by proposing a sound framework for creating new, customized taxonomies of views in a repeatable manner, based on the formal concept of refinement.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Software architecture; Views; Specification; Refinement; Taxonomy; Derivation

1. Introduction

Modern software systems are often so complex that they can be incomprehensible in their entirety. Stakeholders involved in developing such non-trivial systems rely on powerful abstraction mechanisms that present highly specialized information relevant only to their interest. The concept of *views* is one of these abstraction mechanisms and is especially popular in the field of software architecture.

Perry and Wolf [28] consider an *architectural view* as a mechanism to reveal many different facets of a software architecture. Soni et al. [33] show the significance of distinct structures addressed in architectural views as a means of accomplishing separation of concerns. More specifically, IEEE defines an architectural view as a representation of a whole system from the perspective of a related set of concerns [17]. Similar definitions are repeated in the literature as in the case of Bass et al. [4] and Clements et al. [7].

None of these definitions are rigorous enough to precisely define what constitutes a valid or unique view not to mention the relationships among multiple views. Our paper is an attempt to address this lack of rigor by proposing a novel framework for creating more formal taxonomies as a solution. Refinement is the core principle used in developing this framework. We believe that one can extract a conceptual common denominator retaining the most primitive and fundamental building blocks for specification among an infinite set of all the possible views and refer to it as the *baseline view*. The baseline view captures minimal information on a software architecture and plays the role of the origin from which the rest of views are derived through refinement.

The idea of the baseline view is not entirely new and originates from research in architecture interchange languages that provide a generic interchange format for architectural designs. Garlan et al. [12] observe that architecture description methods (especially, languages-based ones) mostly exist in isolation and are not compatible with one another. They propose a language denoted as ACME, featuring a common set of essential specification constructs shared (and minimally required) by any viable language dedicated to describing software architectures. This notion of the least common denominator is certainly applicable to view descriptions and inspired the authors to develop the concept of the baseline view.

Garlan et al., however, do not provide solutions for handling the *excess* architectural information beyond

* Corresponding author. Tel.: +1 913 897 8515; fax: +1 913 897 8681.
E-mail address: saiedian@eecs.ku.edu (H. Saiedian).

0950-5849/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2005.05.006

the scope of ACME, and they mostly ignore it. This paper adopts most of the primitive specification building blocks identified by Garlan et al., but unlike ACME provides a mechanism to harness their expressive power through applying additional constraints so that they can describe the excess architectural information that was not originally describable. The end results of this process are derived views.

The notion of applying constraints to specification constructs to enhance the precision of a specification method has been successfully used in Medvidovic et al.'s [25] research to use a general-purpose modeling language for describing software architectures.

From a practitioner's point of view, the lack of rigor found in the current practice of modelling and specifying archictectural views, means difficulty in adopting the concept of views as a viable software engineering tool. When stakeholders are not able to agree upon (1) whether a specification qualifies to be an architectural view, (2) what differentiates one view from the other, (3) how exactly the constructs of one view are mapped to those of the other, one cannot expect the large-scale use of views to be repeatable any longer.

Repeatability is of great importance in specifying and analyzing views because without it no one will be able to definitively construe the precise meaning of a certain view. Even with an assumption that every one from the same category of a stakeholder community perfectly understands the syntax and semantics of a language used to describe an architectural view, a significant problem of miscommunication potentially exists among people from disparate stakeholder communities using slightly different languages, which leads to inconsistencies between two or more specifications. Repeatability is one of the most critical preconditions for preventing these inconsistencies from occurring.

In fact, part of the problem is that practitioners are often unaware of and overlook these deficiencies, which eventually lead to some serious consequences including ambiguous, inconsistent, and indecipherable view specifications. Not realizing that these negative results stem from inability to properly use the concept of views (but not from inherent flaws in the concept itself), the management may quickly decide that the notion of views is not mature enough for industrial application. After all, some of the most important intended benefits of using views are removing ambiguity, reducing confusion, and facilitating communication among stakeholders. If an argument erupts on whether a class diagram is a detailed design or architectural description (since the use of a class diagram is suggested by Kruchten [21] to document one of his 4+1 views, that is, the logical view), one's confidence toward views evaporates rather quickly. This is especially true in an environment, where multiple people maintain a view, exposing the integrity of a view to the forces of erosion and degradation.

In addition, the problem of the ad hoc view descriptions prevents one from introducing a new view to cope with domain specialization in an efficient and standardized fashion. To incorporate a variety of special, concrete stakeholder concerns that is not easily foreseeable, in diverse problem domains, one must have an extensible taxonomy of views to which a new view can be easily defined and added when necessary. Our framework provides systematic mechanisms for both defining a new view and making it part of an existing taxonomy.

The organization of this paper is as follows. Section 2 reviews the existing taxonomies of views and points out their deficiencies. Section 3 introduces the idea of a baseline view and proposes a refinement-based framework for creating extensible taxonomies of views. Section 3.1 provides a definition of the baseline view. Section 3.2 discusses how the baseline view is refined and how it branches into other unique views. It also describes the criteria that make those derived views unique. Section 4 presents a well-known existing taxonomy in the baseline/derived view framework we propose and examines its consequences. Section 5 provides examples of using the concept of the baseline and derived views to rigorously define views and their relationships. A discussion of further research and concluding remarks in Section 6 rounds out the paper.

2. Related approaches

Currently, several taxonomies of views exist [21,33,14, 18,16,7,29]. Among them, one of the oldest and most well-known is Kruchten's 4+1 view model [21], which is summarized in Fig. 1. A primary criterion used in Kruchten's taxonomy of views is the specific concerns of various stakeholders in a software development project. In his approach, a scenario [20] is a common thread that ties different views together and provides a shared context.

Kruchten advocates the use of a subset of models defined in the Unified Modeling Language (UML) specification [27] as notational mechanisms to document his 4+1 views. For example, he recommends a class diagram to represent the logical view. The 4+1 views are also incorporated into the Rational Unified Process (RUP) [23].

Soni et al. [33] propose a similar set of views. They also have four views denoted as the conceptual (describing architectural structures consisting of major design elements and their relationships), module interconnection (functional decomposition and layers), execution (dynamic run-time

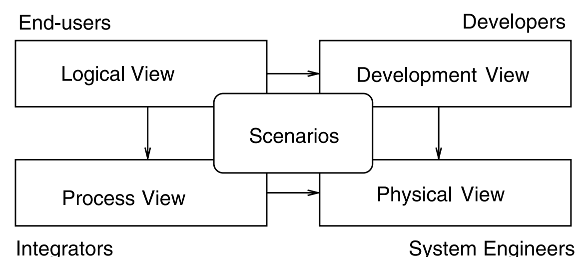

Fig. 1. Kruchten's 4+1 views.

structures), and code views (file structures in a development environment).

More recently, Clements et al. [7] developed their own taxonomy of views that refines Kruchten's 4+1 view concept by means of styles. They categorize views into three major groupings including module, component-and-connector, and allocation. Their categorization is similar with that of Kruchten's except for the fact that they combine physical and development views into the allocation viewtype. Kruchten's process and logical views are comparable to the component-and-connector and module viewtypes.

The mapping from the taxonomy of Kruchten to that of Clements et al. is not done at the same abstraction level. The module viewtype in the Clements et al.'s taxonomy, for example, is just an abstract label referring to a category of views showing how a software system is structured as a set of implementation units [7]. The module viewtype is further refined into concrete views such as the *decomposition*, *uses*, *layered*, and *class* views. The fact that the individual views of Kruchten can be mapped to the abstract categorization of the Clements et al.'s views implies that the latter is more fine-grained than the former.

The core of Clements et al.'s contribution lies in that they map concrete architectural styles to views and establish notational conventions for each style as shown in Fig. 2.

Although more mature than Kruchten's and others, the taxonomy still lacks:

- the rigorous definition of a view,
- the unambiguous criteria used in the classification of views,
- traceability rules governing the ultimate relationships between views belonging to different categories, and
- principles addressing ways to create a new view and relate it to an existing taxonomy.

3. A refinement-based taxonomy of views

In this section, we propose a new kind of taxonomy of views quite unlike conventional classification. It is based on an observation that every view can originate from a single root view denoted as *the baseline* and is the result of continual refinement through the application of more constraints to either the baseline view or other less refined views stemming from the baseline view. Fig. 3 visualizes the refinement-based taxonomy of views.

3.1. A definition of the baseline view

Each view records a snapshot of decisions made by an architect. The nature of these captured decisions, when considered in terms of the degree of their uniqueness, can range from highly esoteric to extremely common.

For example, it is possible that one invents a completely new architectural mechanism of which no one is aware. On the other hand, an architecture of one's choice can turn out to be fairly mundane. Architectures using well-known patterns [32] (such as client–server, pipe-and-filter, etc.) belong to this latter category.

In the same context, there are a set of views that tend to be more open-ended and capable of supporting almost any type of architectural decision-making while there are those limited to capturing only certain architectural decisions. For the simplicity of our discussion in this paper, we refer to the former as a *generic* view and the latter as a *derived* view.

Kruchten's 4+1 views are excellent examples of dominantly generic views. Regardless of the problem domain and its scale, the 4+1 views are nearly always relevant. After all, one can hardly imagine a software-intensive system without its logical, physical, process, or development aspect. Table 1 provides short definitions for Kruchten's 4+1 views.

Compared with Kruchten's 4+1 views, Clements et al.'s views are more fine-grained and refined. Some of their views (especially, the views belonging to the component-and-connector view type) even enforce architectural styles, thereby limiting the kinds of architectures specifiable in them. A view meant for architectures conforming to the pipe-and-filter style is most likely ill-suited for describing a client–server architecture (Fig. 2).

Medvidovic et al. [25] report a similar trend in their discussion of Architecture Description Languages (ADLs) [26]. They point out that an ADL such as Wright [3] does not enforce the rules of a particular style while another ADL denoted as C2 [24] does so.

Note that one cannot invariably claim that a view is absolutely generic or derived. In other words, the terms (both generic and derived) have a strong connotation of being relative. Therefore, it is theoretically possible to set up

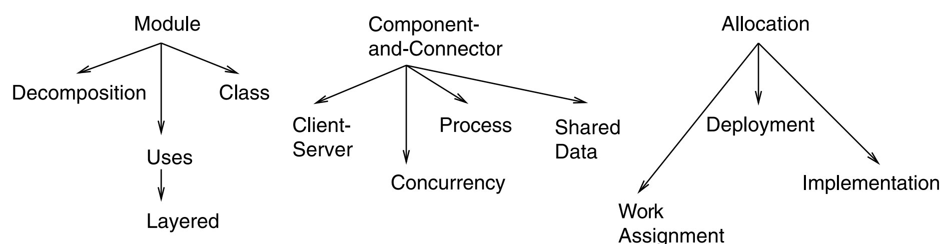

Fig. 2. Clements et al.'s taxonomy of views.

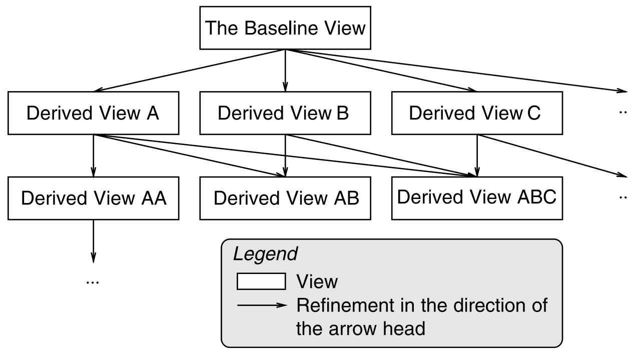

Fig. 3. The refinement-based taxonomy of views.

the most generic baseline (hence the name baseline view) and define derived views relative to the baseline view.

For the past decade, researchers strove to identify a common denominator for all the existing ADLs since they believed that it would enable the exchange of specifications among disparate ADLs. ACME [12] is the direct result of this effort. Additional efforts are made by Dashofy et al. [9,10] to extend the ACME research and develop an eXtensible Mark-up Language [6] (XML)-based ADL (xADL). Both ACME and xADL provide excellent ideas on what could be the bare minimum set of constructs in specifying software architectures and serve as a basis to implement the baseline view concept introduced earlier.

Components and connectors are the most important building blocks in ACME and xADL. Thus, we define the most fundamental constructs in the baseline view as components and connectors. We also let both components and connectors have zero or more ports that are interfaces to other components and connectors.

Components, connectors, and ports by themselves are not sufficient to specify even the most rudimentary software architecture. One still needs to define the mappings between component and connector ports, and configuration plays this role.

Constructs described so far are static in nature, and they are meant to retain structural information. To specify the behavioral aspect of a software architecture, one needs, at a minimum, ways to specify events and a sequence in which the events occur [11]. This paper adopts the connector-oriented specification approach [2,1] that advocates tying the behavioral aspect of a software architecture to connectors. We refer to the behavioral specification (involving events and sequences) in a connector as a *protocol*. We provide the detailed definition of each baseline view construct in Table 2.

Table 1
Definitions for Kruchten's 4+1 views

View name	Definition
Logical	Shows the correspondence between functional requirements and high-level conceptual solutions
Physical	Describes the mapping(s) of the software onto the hardware and reflects its distributed aspect
Process	Captures the concurrency and synchronization aspects of the design
Development	Describes the static organization of the software in its development environment

Table 2
Definitions of the baseline view constructs

Construct	Definition
Component	A locus of computation or data storage
Connector	An abstraction of communication or interactions between two or more components
Port	A directional interface (in, out, or inout) between a component and a connector
Configuration	The attachments of components and connectors to eventually form an architectural topology
Protocol	Dynamic interactions between two or more components specified using events occurring in prescribed sequences

In summary, the baseline view embodies the minimal requirements for any valid views. Its relation to derived views is similar to that of an abstract class to the corresponding concrete descendants in an object-oriented design specification. For example, an abstract class provides a common conceptual base for its specialized sub-classes and is never instantiated. Analogously, the baseline view acts as the origin of any arbitrary derived view and is not directly used to specify a stakeholder-specific architectural view. It is a logical reference point whose existence is essential for unambiguously and compactly defining other views by avoiding repeating the same definitions of constructs and constraints. A formal definition of the baseline view is provided in Section 5.

3.2. Refinement of the baseline view and different dimensions involved in making an architectural view unique

From the baseline view proposed in Section 3.1, one can create derived views by applying additional constraints to one or more of the modeling constructs of the baseline view. For example, a plain *layered* view (an architectural style mapped to the module view category of the Clements et al.'s taxonomy of views in Fig. 2) can be derived from the baseline view by adding the following constraints:

- For components, c1 and c2 communicating through a connector, the communication is always uni-directional, especially in the direction of an upper layer component c1 toward either the same or its directly lower layer counterpart, c2.

Therefore, components in the layered view conform to the *anti-symmetry* rule which forces an upper layer component to always use a component belonging to the same or a lower layer, but not the other way around.

One can also replace this natural language description of constraints with one specified in first order predicate logic

expressions (as demonstrated in our example in Section 5) or a formal language such as Z [19].

A derived view can be further specialized by imposing more constraints, implying that derived views can be created not only from the baseline view, but also from one or more other derived views. Using this mechanism of derivation, one can create an infinite number of views tailored for his or her own context of architectural specification.

Many researchers propose their own set of views (for instance, those discussed in Section 3). Despite the differences in the actual presentation of these views, they all share relatively common criteria to categorize their views. Although few of them formally define relationships among different kinds of views in terms of the baseline view and its derived views, one can still use some of these criteria to develop constraints that can be used for creating derived views. This paper looks into them in the following sections.

3.2.1. Disciplines in software development

Generally speaking, a software application goes through multiple iterations of several well-defined disciplines throughout its lifetime (regardless of methodologies used to develop it, such as Rational Unified Process [22], the spiral model [5], the waterfall model [31], etc.). Although there exist differences in their nomenclature depending on the development life cycle methodology of one's choice, these disciplines are classified largely into requirements, analysis, design, implementation, testing, customer-support, configuration/change management, project management, etc.

Each of these disciplines represents a stakeholder who, in turn, demands corresponding architectural views. The information conveyed in these views is mostly about the artifacts of a given discipline. For instance, those participating in a production support stakeholder role (in the context of an industrial strength software development environment) can greatly benefit from views such as deployment or implementation in Fig. 2. This is because they are responsible for deploying code, monitoring and responding to issues, and ultimately finding resolutions. The artifacts addressed in the deployment view include deployment scripts, property files, configuration files, processes, threads, etc. The implementation view is associated with source code, property files, configuration files, etc. Note that the two views sometimes describe the same artifacts but in different contexts.

3.2.2. Orientation toward static vs. dynamic nature of software

The structural aspect of a software architecture often affects the runtime behavior of software. In this case, it makes little sense to make a definite statement on whether a view is strictly static or dynamic. As a result, one can only decide the overall orientation of a view toward either the static or dynamic aspect of a software architecture.

However, it is possible to distinguish views involving runtime components and connectors from those associated with non-rune time elements. Therefore, in this paper, views are referred to as dynamic when they only contain runtime architectural elements. Otherwise, they are labeled as static.

3.2.3. Quality attributes

Views can be used to verify and validate whether a software architecture conforms to a certain quality requirement. The development view in Fig. 1, for example, helps reasoning about software reuse, portability, and security while the physical view reveals a system's conformance to availability, reliability, performance, and scalability [8].

3.2.4. The domain

Certain views become more useful in a highly specialized domain. Views highlighting the concurrency and parallelism of a system are beneficial to the development of a distributed system because they can help spot a problem such as a deadlock. Beside the generic views applicable to an arbitrary domain, more specialized (customized) views can enhance the ability of stakeholders to ensure the satisfactory implementation of a proposed software architecture.

3.2.5. The scope and granularity

The fact that views implement the notion of separation of concerns implies that a view has an inherent tendency to avoid covering the entirety of a software architecture. This idea of intentional incompleteness can be further examined in terms of two orthogonal concepts: scope and granularity. Scope dictates the breadth of a view. We define the breadth as the ratio of the number of depicted components (as defined in Section 3.1) in a given view to all the known components of a software architecture at an abstraction level to which the view pertains. More precisely,

$$S = \frac{N_{cv}}{N_{ca}}$$

where S, N_{cv}, and N_{ca} stand for the scope of a view, the number of components found in a view, and the number of all the known components. The scope of a view is referred to as *monolithic* when $S=1$. It is *partial* when $S<1$. While scope represents the coverage aspect of a view, granularity deals with its level of abstraction. In this paper the granularity of a view is defined as:

$$G = \frac{N_{cam}}{N_{cah}}$$

where G and N_{cah} are granularity and the number of all the known components when the view is monolithic. N_{cam} is the minimum value among the available numbers of all the known components. The maximum value of granularity is 1. The value becomes smaller as one drills down to more details of a software architecture and describes them in a view. Here, we make an assumption that a view has at least one component.

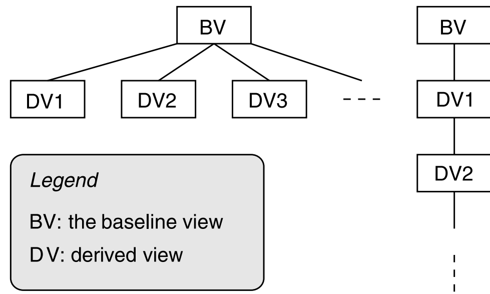

Fig. 4. Unnecessarily flat or steep taxonomies of views.

4. Specifics of the new taxonomy

In principle, the new taxonomy described so far does not dictate where in its hierarchy each derived view must fit. Neither does it prescribe the details of the views. Rather, it is a *meta*-taxonomy that gives a rise to new taxonomies conforming to a paradigm built upon concepts such as the baseline, derived views, refinement, and constraints (all explained in Sections 3, 3.1, and 3.2). The details necessary to form a tangible taxonomy are left up to a development organization that adopts the meta-taxonomy. The emphasis on macro-scale factors embodied in the meta-taxonomy is due to our finding that the usefulness of a view lies in how customizable it is for the need of those interested. There is, of course, the danger that this flexibility might be abused. For example, an inexperienced user can come up with an unnecessarily flat or steep taxonomy as shown in Fig. 4.

Here we revisit the views of the Clement's et al.'s taxonomy [7] and redefine them in the framework of our meta-taxonomy to:

- provide a standard procedure through which new domain-specific taxonomies can be created,
- make a ready-made set of views available for those willing to follow the principles of our meta-taxonomy, but not wanting to develop their own taxonomy, and
- discourage one from formulating the ill-formed taxonomies depicted in Fig. 4.

We regard a taxonomy as the consequence of the repeated creation of views in a systematic way. The same train of thought leads us to a conclusion that the distinctive specifics of different taxonomies are the results of applying a set of constraints (discussed in Section 3.2) to the baseline view and its descendants in a certain sequence. Note that what decides the overall topology of a taxonomy and the contents of its constituent views is not only the type of the constraints, but also the order in which they are applied.

Our effort to redesign the Clements et al.'s taxonomy takes full advantage of these ideas. As a result, each view is scrutinized concerning:

- the required constraints to refine the baseline view into the derived view under scrutiny and
- a sequence in which these constraints are imposed.

The first constraint we use is whether all the architectural elements in a view are either *logical* or *physical*. Typically identified by the design discipline, the logical elements are defined as the conceptual (initially existing only in the imagination of stakeholders) abstractions of a desired system structure and its behavior. On the contrary, the physical elements are the abstractions of the tangible implementations of the logical elements. They are produced during the implementation discipline and directly traceable to concrete objects in the real world, such as source code snippets, application servers, self-contained software components, etc.

Static versus dynamic is the second constraint. The static elements of a view distill only the topology of an architecture while their dynamic counterparts represent the run-time behavior projected on top of the static elements.

With these two basic constraints, one can come up with the four unique derived views shown in Fig. 5.

Views belonging to the module view type in the Clements et al.'s taxonomy can be placed under the physical:static node in the taxonomy shown in Fig. 5 since modules in these views are defined as *implementation* units of software [7]. To form the decomposition view, one needs to refine the constructs of the baseline view using the constraints summarized in Table 3 in addition to those of physical: static. We give distinctive names to the refined components and connectors for a convenient reference. The descriptions of the constraints for this and ensuring views are adapted from Clements et al. [7]. In a similar fashion, the uses view can be derived and its constraints are summarized in Table 4.

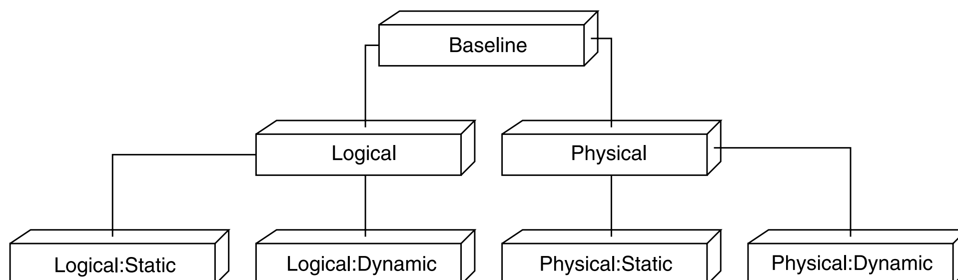

Fig. 5. The four derived views.

Table 3
Additional constraints for the decomposition view

Refined baselines construct	New name	Constraints
Component	Module	N/A
Connector	Decompo-sition	A part/whole relationship between the submodule A-the part and the aggregation module B-the whole
Configur-ation	N/A	No loops are allowed. A module cannot be part of more than one module

The layered view can be refined from the uses view by adding the constraints shown in Table 12.

Finally, the generalization view is formed from the physical:static view in Fig. 5 with the additional constraints in Table 5.

Views falling into the category of the component-and-connector view type can be associated with the physical: dynamic node in Fig. 5 because the types of components and connectors involved have run-time presence such as processes, objects, clients, servers, and data stores [7]. Table 6 shows assumed constraints in the pipe-and-filter view.

One can formally specify the protocol of a pipe using an ADL like Wright [2,1] as shown below (adapted from Medvidovic et al. [25]):

connector Pipe =
role Writer = write → Writer ⊓ close → √
role Reader =
let ExitOnly = close → √
in let DoRead = (read → Reader □ read-eof → ExitOnly)
in DoRead ⊓ ExitOnly
glue = let ReadOnly = Reader.read → ReadOnly □ Reader.read-eof → Reader.close → √ □ Reader.close → √
in let WriteOnly = Writer.write → WriteOnly □ Writer.close → √
in Writer.write → glue □ Reader.read → glue □ Writer.close → ReadOnly □ Reader.close → WriteOnly

Wright is an ADL specializing in describing dynamic structures involved in a software architecture in terms of behavioral characteristics of components and their interactions. In Wright, interactions between components are embodied in an independent specification construct (namely, a connector). Both components and connectors are associated with a type that has its own instances.

Table 4
Additional constraints for the uses view

Refined base-line construct	New name	Constraints
Component	Module	N/A
Connector	Uses	Module A uses module B if A depends on the presence of a correctly functioning B in order to satisfy its own requirements

Table 5
Additional constraints for the generalization view

Refined baseline construct	New name	Constraints
Component	Module	N/A
Connector	Generalization	An *is-a* relationship can be established between two participating components

Wright uses a subset of communicating sequential processes (CSP) [15], a formal language developed by Hoare to model concurrency. CSP uses notations such as □, ⊓ and √ to represent state transitions (deterministic choice, non-deterministic choice, and a successful event). For instance, the expression

$$\text{Writer} = \text{write} \rightarrow \text{Writer} \sqcap \text{close} \rightarrow \surd$$

states that Writer either keeps writing or closes the pipe, and the decision on whether it needs to continue writing or to stop is made non-deterministically.

The role sections of the specification describe the dynamic behavior of two different types of filters (namely, those reading and writing) in isolation, which is useful, in addition to the static constraints for components in Table 6, for identifying components compatible with the pipe-and-filter view. The glue specification presents scenarios (summarized by the UML state transition diagram in Fig. 6 adapted from Medvidovic et al. [25] on how a pipe accommodates the interactions of filters and how pipes and filters work together in general.

In the figure, Reader.read represents an event that triggers a Reader to start reading while Writer.write is another event making a Writer begin writing. There are also other events closing a pipe for either reader or writer. From an unknown state, a transition to states labeled as ReadOnly, WriteOnly, and glue deterministically occurs depending on what event (namely, read, write, and close) precedes the state transition.

Table 6
Additional constraints for the pipe-and-filter view

Refined base-line construct	New name	Constraints
Component	Filter	A component has at least one input port or one output port. It reads streams of data from its input port, transforms them, and writes them to its output port.
Connector	Pipe	A connector has one input port and one output port. It conveys streams of data from one filter to another.
Port	N/A	The directionality of ports are either in or out.
Configuration	N/A	The out ports of filters are connected to the in ports of pipes. The in ports of filters are connected to the out ports of pipes.

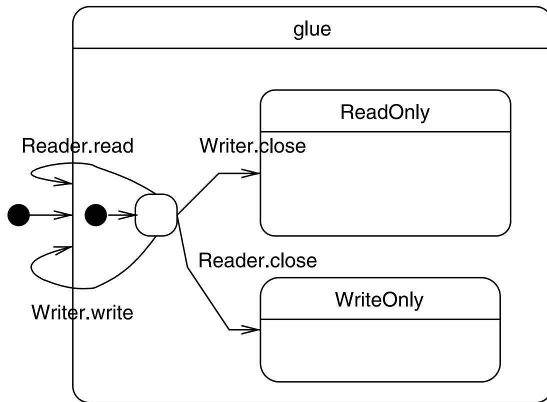

Fig. 6. UML state transition diagram of the pipe connector.

Some of the scenarios depicted in Fig. 6 include:

- one filter reading while another writing,
- one filter reading while another ceasing to write,
- one filter writing while another ceasing to read, etc.

As one adds more scenarios to a protocol, the affected view becomes more refined, subsequently resulting in more derived views. We realize that the static structural constraints found in the pipe-and-filter view (Table 6) are sufficient to distinguish it from other views in Fig. 2. Of course, the protocols dictating the behavior of pipes are what distances the pipe-and-filter view further away from the rest of non-pipe-and-filter-related views. As a result, we do not discuss the details of protocols in the remaining descriptions of views classified as part of the component-and-connector view type unless the structural constraints fail to make them unique.

Another view belonging to the component-and-connector view type is the peer-to-peer view. Client–and–server view can be derived from the peer-to-peer view (Table 7) by adding the constraints in Tables 8.

Regarding the constraint for the configuration of the client–server view, note that this constraint is applied to the configuration instead of a component. If the same constraint is applied to a component, the affected component will be restricted to assuming only a single role at a time, which is not realistic (what if a component acts as a server on one side and behaves as a client on the other?). Configuration identifies an interface of a component (not the entire component) as a server or client. Since a component can have multiple interfaces, it can assume multiple roles (both server and client).

Table 7
Additional constraints for the peer-to-peer view

Refined baseline construct	New name	Constraints
Component	Peer	Components are code units (for example, classes or a set of classes)
Protocol	N/A	One component can invoke the service of the other

Table 8
Additional constraints for the client–server view

Refined baseline construct	New name	Constraints
Component	Clients and Servers	N/A
Configuration	N/A	Once designated as a server (or a client), a component cannot change its role
Protocol	N/A	Servers are components providing a service. Clients are components requesting a service. Only a client can invoke the service of a server

When the roles of a client and a server are, respectively, limited to requesting and providing services to create, read, update, and delete data on a persistent platform, the client–server view is further refined into a primitive type of the shared-data view. These specialized clients and servers are referred to as shared-data accessors and repositories. A more sophisticated variety of the shared-data view enriches the protocol of the basic shared-data view by allowing changes in the repositories to trigger a wide range of services (including requesting the data-centric operations mentioned above) from accessors.

The communicating processes view also belongs to the component-and-connector view type. Table 9 lists the constraints relevant to the communicating process view.

The last we discuss in the component-connector view type is the publish-subscribe view whose constraints include those in Table 10.

The views that are part of the allocation view type can be connected to the physical:static node in Fig. 5 since:

- the implementation view describes the correlation between modules and a file system containing the modules,
- the work assignment view describes the correlation between modules and (human) resources responsible for the development of the modules, and
- the deployment view describes the correlation between the concurrent units of processing (usually processes)

Table 9
Additional constraints for the communicating-processes view

Refined baseline construct	New name	Constraints
Component	N/A	Components are the *concurrent* units of processing including processes and threads
Protocol	N/A	Components communicate with each other. Communicating is more than simply requesting and providing services. It also includes activities such as exchanging data, passing messages, synchronization, control, etc

Table 10
Additional constraints for the publish-subscribe view

Refined baseline construct	New name	Constraints
Component	Publisher/subscriber	N/A
Connector	Message/event bus	N/A
Protocol	N/A	Components designated as publishers send a category of messages/events to a centralized connector, which eventually forwards them to other subscribed components

and the environmental elements of varying degree of granularity.

We assume that the correlations involved in each view are inherently static and specify the arbitrary configurations of components and connectors. Therefore, the existence of dynamic components (the concurrent units of processing) in the deployment view is an insignificant factor in deciding whether an entire view is static or dynamic. However, the deployment view may also be categorized as physical: dynamic once the correlation itself is the function of time or other external influences. In this case, it is necessary to specify the protocol of the affected view.

Fig. 11 depicts the final taxonomy, which is apparently skewed. That is, the logical:static and logical:dynamic views do not have any children. What we conclude from this observation is that the Clement's et al.'s taxonomy either lacks the logical views in general or simply ignores the boundaries between physical and logical views, therefore, using physical views to describe their logical counterparts, too.

5. Examples

In this section, we present tangible examples showing how a real-life software development organization can benefit from our meta-taxonomy for architectural views to establish its own domain-specific taxonomy complete with a clear definition for each unique view type involved and its relationships with others.

The organization in our example is composed of stakeholders participating in development efforts compliant with the Java 2 Platform, Enterprise Edition (J2EE) standard [34]. J2EE defines a standard architecture for a platform used in developing component-based, multi-tier enterprise information systems (EISs).

The main purpose of the J2EE platform is to reduce complexity involved in EIS development by relying on standardized, modular components called Enterprise Java Beans (EJBs) and by automatically providing services to ensure reliability, scalability, security, and availability without requiring complex programing from its users.

The major J2EE stakeholders and the details of their roles are summarized in Table 13.

Fig. 7 (adapted from [33]) depicts an ad hoc view of the J2EE platform to which many of software practitioners are accustomed through numerous meetings, where they draw boxes and lines in their impromptu discussions of a software architecture.

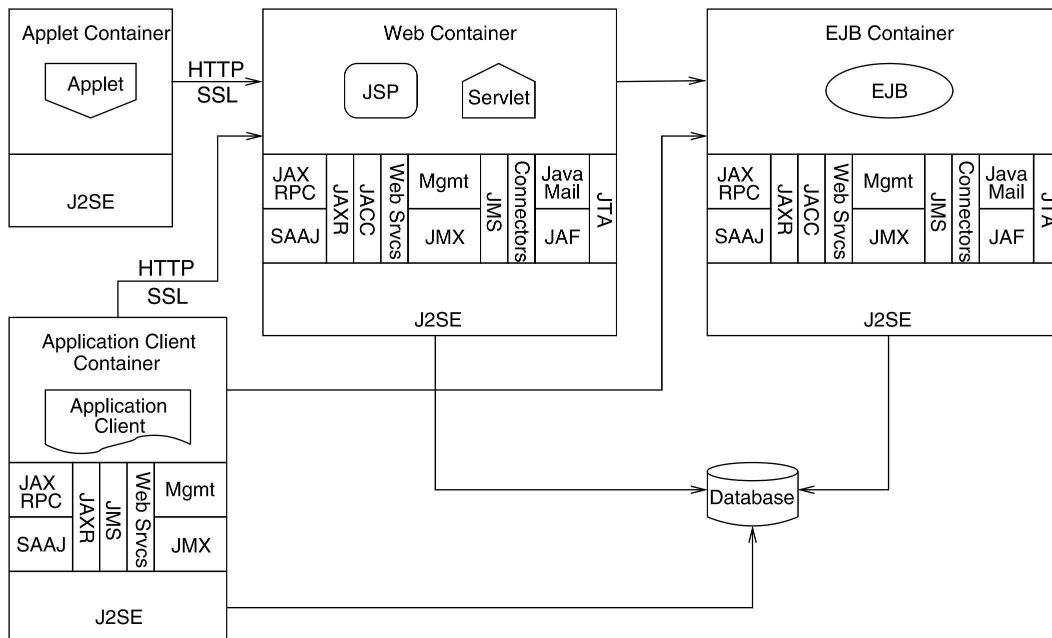

Fig. 7. An ad hoc view of the J2EE platform.

The ultimate goal in these exercises is to provide a quick overview of a software architecture under scrutiny. This type of view is effective in a sense that it satisfies the need of conveying ideas in an intuitive manner. However, the view is inherently ambiguous because there is an ample room for personal interpretation even after an explanation is given by the author himself or herself. The ambiguity worsens when only the graphical form of the view is documented leaving out the original intentions of its creator, which is commonly the case.

If there exists a systematic, repeatable standard of unambiguously communicating what constitutes a unique view and its semantic nuances, one can alleviate the problem of misinterpretation because the standard tells how to interpret a view. In other words, a view specification accompanied by a view definition provides helpful information to understand a view description and greatly reduces the probability of misunderstanding.

This is exactly what our meta-taxonomy of views is for: providing a compact definition of a view. Once a stable set of views frequently used for an organization emerges, the definition can be kept at a single location and maintained by a single authority. The presenter of a view can simply refer to its name, and the audience is advised to consult the definition.

Assume that a common concern across all the stakeholders in Table 13 is identifying logical components and the static *uses* dependency relationships between them. For this goal, one may define a view (V_{uses_def}) as follows.

First, we formally define the baseline view:

$$S_{\text{csrt}} = \{C_{\text{omp}}, C_{\text{onn}}, C_{\text{onf}}, P_{\text{ort}}, P_{\text{rot}}\} \tag{1}$$

where S_{csrt} is composed of specification constructs, C_{omp}, C_{onn}, C_{onf}, P_{ort}, and P_{rot}, which, respectively, represent sets of components, connectors, configurations, ports, and protocols as defined in Section 3.1.

Expressions (2)–(7) are a minimum set of constraints imposed upon S_{csrt} and necessary to define the baseline view.

$$C_b1 : \forall x \in C_{omp} \bullet \exists\, y \in P_{ort} \bullet has(x, y, 0) \tag{2}$$

$$C_b2 : \forall x \in C_{onn} \bullet \exists\, y \in P_{ort} \bullet has(x, y, 2) \tag{3}$$

$$C_b3 : \forall x \in P_{ort} \bullet in_direction(x) \vee out_direction(x) \vee inout_direction(x) \tag{4}$$

$$C_b4 : \forall x \in C_{omp} \bullet static(x) \Rightarrow \neg \exists\, y \in C_{onn} \bullet dynamic(y) \tag{5}$$

$$C_b5 : has(x,y) : x \in C_{onn} \wedge y \in P_{rot} \bullet has\cdot(x,y) \Rightarrow \forall x \bullet \forall z \in C_{omp} \bullet dynamic(x) \wedge dynamic(z) \tag{6}$$

$$C_b6 : \forall x,y \in P_{ort} \bullet \exists\, !z \in C_{onf} \bullet mapped_to_by(x,y,z) \tag{7}$$

$$C_b7 : \forall x \in V \bullet \exists\, !y \in C_{onf} \bullet has(x,y) \tag{8}$$

They state that:

- each component has zero or more ports (2),
- each connector has two or more ports (3),
- the direction of each connector communication/interaction is in, out, or inout (4),
- an explicit connector exists only when all the components in a view are dynamic (5).
- a protocol is used only when all the components and connectors in a view are dynamic (6),
- one port is mapped to the other by a unique configuration (7), and
- each view has only one configuration (8).

Therefore, the baseline view definition (B_{def}) consists of specification constructs (S_{csrt}) and constraints (C_b) as in expressions, (9) and (10).

$$C_b = \{C_b1, \ldots, C_b7\} \tag{9}$$

$$B_{def} = (S_{csrt}, C_b) \tag{10}$$

Now, every time one wants to define a new view, he or she can reuse this baseline view definition as in expression (11). A new view is defined in terms of the baseline view and a set of additional constraints (C_d). In this case, we define a logical: static: uses view (V_{uses_def}). Expression (12) simply states that there is no common constraint between the baseline view and the newly derived view.

$$V_{uses_def} = (B_{def}, C_d) \tag{11}$$

$$C_b \cap C_d = \varnothing \tag{12}$$

C_d consists of:

$$C_d1 : \forall x \in C_{omp} \bullet \text{log}ical(x) \tag{13}$$

$$C_d2 : \forall x \in C_{omp} \bullet static(x) \tag{14}$$

$$C_d3 : \forall x \in C_{omp} \bullet \exists\, y \in C_{omp} \bullet (uses(x,y) \oplus uses(y,x) \wedge (x \neq y) \tag{15}$$

and $C_d = \{C_d1, C_d2, C_d3\}$, where *logical*($x$) and *static*($x$) are predicate logic expressions that become true when x is a logical and static specification construct belonging to S_{csrt}. Expressions, (13) and (14) state that all the components are logical and static. *uses*(x, y) is a predicate logic expression that becomes true when x depends on the presence of a correctly functioning y in order to satisfy its own requirements. Expression (15) states that every component either uses or is used by every other component and that a component is not allowed to use itself.

When V_{uses_def} is applied to the ad hoc view in Fig. 7, a new structure emerges as shown in Fig. 8. A summary of notations used in the figure is provided in Table 11. To simplify the drawing, we limited the scope ($S < 1$ according

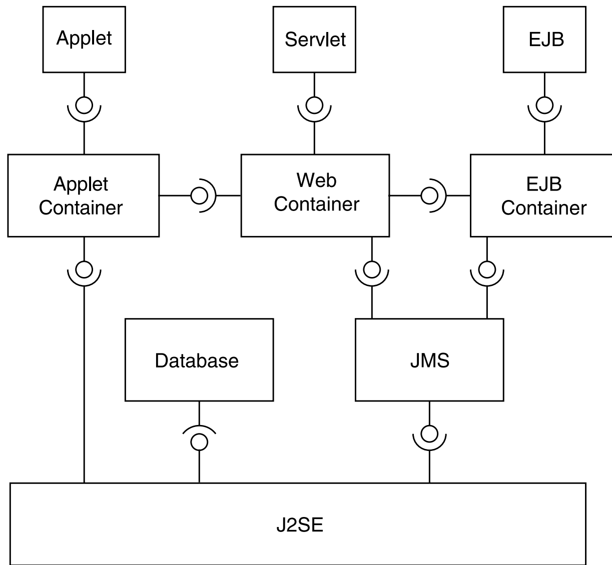

Fig. 8. The logical:static:uses view.

to Section 3.2.5) of our view so that we consider only the Java messaging service (JMS) [34] standard of the J2EE platform out of many such as JTA, JAXR, etc. (as shown in Fig. 7). JMS is a standard service provided by a J2EE container that allows asynchronous communication between components.

Note that expressions, (11)–(15) concisely and precisely define the view and remove the ambiguity associated with the ad hoc view. The relationship between the new view and the baseline view is given in expression (11).

As discussed in Section 4, another common view denoted as the *layered* view (Table 12) can be derived from the *uses* view. In Clements et al.'s taxonomy, both the uses and layered views are physical: static, but in this example, we allow the logical: static versions of them.

The layered view definition $V_{layered_def}$ is given using the expressions (16)–(19).

$$V_{\mathrm{layered_def}} = (B_{\mathrm{def}}, C_{\mathrm{d_uses}}, C_{\mathrm{d_layered}}) \tag{16}$$

$$C_{\mathrm{b}} \cap C_{\mathrm{d_uses}} \cap C_{\mathrm{d_layered}} = \varnothing \tag{17}$$

C_{d_uses} is equivalent to C_d in expressions (11) and (12). $C_{d_layered}$ contains:

Table 11
The notations in Fig. 8 and their meanings

Notation	Meaning
□	Component
⧄	Connector
—○	Port for an outbound communication/interaction
—(	Port for an inbound communication/interaction
—○—	Ports participating in a duplex communication/interaction

Table 12
Additional constraints for the layered view

Refined baseline construct	New name	Constraints
Component	Layer	N/A
Connector	Allowed to Use	N/A
Configuration	N/A	Anti-symmetry (If layer A uses layer B, layer B cannot use layer A and is the same or lower layer of layer A.)

$$C_{\mathrm{d_layered}}1\ \forall x \in C_{\mathrm{omp}} \bullet \exists!k \in \{\mathbb{N}\} \bullet \mathrm{layer}(x) = k \tag{18}$$

$$C_{\mathrm{d_layered}}2\ \forall x,y \in C_{\mathrm{omp}} \bullet \mathrm{uses}(x,y) \Rightarrow \mathrm{layer}(x) \geq \mathrm{layer}(y) \tag{19}$$

and $C_{d_layered} = \{C_{d_layered}1,\ C_{d_layered}2\}$. Expression (18) states that each component is defined in exactly one layer while (19) says that each component uses others situated only on the same layer or lower layers. When applied to the logical: static: uses view in Fig. 8, $V_{layered_def}$ makes another architectural structure surface as shown in Fig. 9.

Here we provide the natural language definitions of the predicates. One can add more rigor by formally defining them. In addition, depending on the domain, different definitions may exist for the same predicate.

Although semantically far different, the formal specification of the implementation view in the Clements et al.'s taxonomy is very similar with that of the logical: static: uses

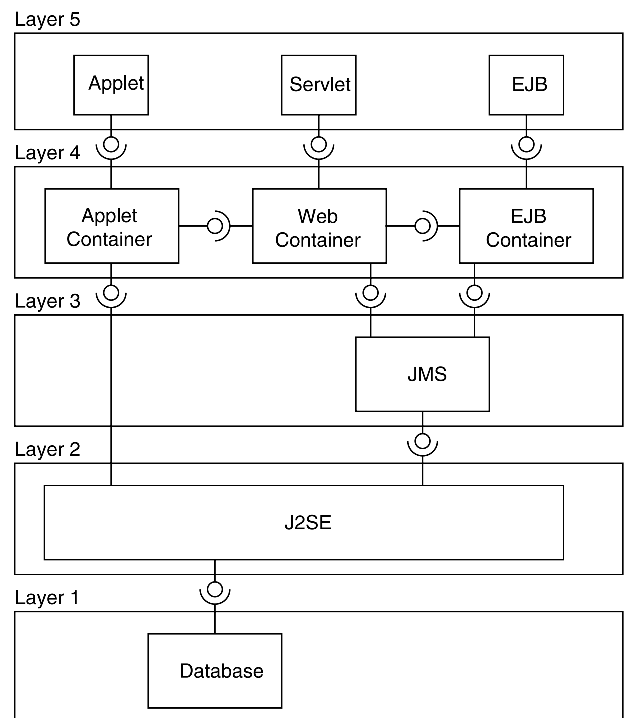

Fig. 9. The logical:static:layered view.

view as shown below.

$$V_{imp_def} = (B_{def}, C_{d_imp}) \quad (20)$$

$$C_b \cap C_{d_imp} = \varnothing \quad (21)$$

C_{d_imp} consists of:

$$C_{d_imp}1: \forall x \in C_{omp} \bullet module(x) \oplus file(x) \quad (22)$$

$$C_{d_imp}2: \forall x \in C_{omp} \bullet physical(x) \wedge static(x) \quad (23)$$

$$C_{d_imp}3: \forall x \in C_{omp} \bullet \exists y \in C_{omp} \bullet (contains(x,y) \oplus contains(y,x)) \wedge (x \neq y) \quad (24)$$

$$C_{d_imp}4: \forall x,y \in C_{omp} \bullet contains(x,y) \Rightarrow module(x) \wedge file(y) \quad (25)$$

and $C_{d_imp} = \{C_{d_imp}1,\ldots,C_{d_imp}4\}$.

Expressions ((22)–(25)) state that:

- a component is either a module or file (22),
- all the components are physical:static (23),
- every component either contains or is contained by every other component (24),
- a component is not allowed to contain itself (24), and
- a file must contain a module, not the other way around (25).

In addition to the general concerns addressed by the logical:static:uses, logical:static:layered, and physical:static:implementation views, an individual stakeholder in Table 13 needs more specialized views capturing his or her own unique set of concerns. For example, an application assembler wants to know what aspect of J2EE he or she has to deal with to fulfill his or her responsibilities. One can develop a customized view for this purpose as specified in expressions (26)–(39).

$$V_{asb_def} = (B_{def}, C_{d_asb}) \quad (26)$$

$$C_b \cap C_{d_asb} = \varnothing \quad (27)$$

C_{d_asb} consists of:

$$C_{d_asb}1: \forall x \in C_{omp}, y \in C_{onn} \bullet logical(x) \wedge logical(y) \quad (28)$$

$$C_{d_asb}2: \forall x \in C_{omp}, y \in C_{onn} \bullet dynamic(x) \wedge dynamic(y) \quad (29)$$

$$C_{d_asb}3: \forall x \in C_{omp} \bullet client(x) \oplus ejb(x) \quad (30)$$

$$C_{d_asb}4: \forall x \in C_{onn} \bullet ejb_container(x) \quad (31)$$

$$C_{d_asb}5: \forall x \in C_{omp} \bullet ejb(x) \Rightarrow session_bean(x) \oplus entity_bean(x) \oplus message_driven_bean(x) \quad (32)$$

Table 13
The major J2EE stakeholders and the details of their roles (adapted form Roman et al. [30])

Stakeholder	Description
Bean provider	The role of a bean provider is supplying reusable, commercial off-the-shelf software components called *enterprise beans*, especially designed for the use in a J2EE environment. Enterprise beans are not complete business solutions, but can be deployed and assembled to form a complete application.
Application assembler	The application assembler figures out what enterprise beans are necessary to build a specific business solution and how they fit together. The role involves developing glue code that combines enterprise beans and writing domain-specific enterprise beans. The application assembler is the consumer of the enterprise beans produced by the bean provider role.
EJB deployer	After the application assembler finishes developing a new application, the EJB deployer deploys the source code into the production environment. The responsibilities of this stakeholder include: (1) resolving environment issues associated with deploying a J2EE application, (2) identifying application flaws and separating them from the environment problems, (3) ensuring security, (4) providing suggestions for purchasing new hardware to improve performance, etc.
System administrator	Administrators maintain the J2EE platform and software components installed on it, and monitor them for any problems in the production environment.
Container provider	The container provider supplies an implementation of the J2EE platform standard. The platform acts as a middle-ware and provides a means to manage and access the enterprise beans.

$$C_{d_asb}6: connected(x,y) \Rightarrow x,y \in Z \quad (33)$$

$$C_{d_asb}7: \forall x \in C_{omp} \bullet \exists ! y \in R \bullet is_home(y) \quad (34)$$

$$C_{d_asb}8: \forall x \in C_{omp} \bullet \exists ! y \in R \bullet is_remote(y) \quad (35)$$

$$C_{d_asb}9: \forall x \in C_{omp} \bullet \exists ! y \in R \bullet is_context(y) \quad (36)$$

$$iC_{d_asb}10: \forall x \in C_{onn} \bullet \exists ! y \in R \bullet is_home(y) \quad (37)$$

$$C_{d_asb}11: \forall x \in C_{onn} \bullet \exists ! y \in R \bullet is_remote(y) \quad (38)$$

$$C_{d_asb}12: \forall x \in C_{onn} \bullet \exists ! y \in R \bullet is_context(y) \quad (39)$$

and $C_{d_asb} = \{C_{d_asb}1,\ldots,C_{d_asb}6\}$.

Expressions (28)–(39) state that:

- all the components and connectors are logical (28),
- all the components and connectors are dynamic (29),
- the components are either clients or EJBs (30),
- all the connectors are EJB containers (31),
- EJB is session, entity, or message-driven bean (32),
- Only two ports of the same type can be connected (33),
- A client has one port whose type is Home (34),
- A client has one port whose type is Remote (35),
- A client has one port whose type is Context (36),

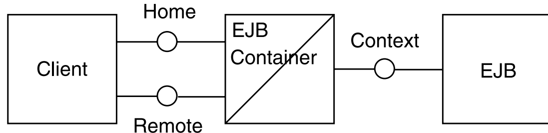

Fig. 10. A customized view for an application assembler.

- An EJB container has one port whose type is Home (37),
- An EJB container has one port whose type is Remote (38), and
- An EJB container has one port whose type is Context (39).

Since the view is dynamic (expression (29)), protocols for the EJB container must be specified. For example, the following interactions occur every time the use of an EJB is requested by a client:

- the EJB container instantiates the requested bean,
- the EJB container gives a context to the bean,
- the bean stores the context for later use,
- the EJB container executes the business logic on the bean, and
- the bean uses the context if necessary.

The context is a gateway a bean can use to access the environment information and limited behavior of an EJB container. One of the simplest instances of V_{asb_def} is depicted in Fig. 10.

In this section, the authors have provided concrete examples demonstrating how one can apply the baseline and derived view concepts to rigorously defining architectural views and specifying their instances. From the examples it is apparent that one can further improve the clarity of a view definition by formally defining the axioms (for instance, the definition of *uses*(x, y)) presented in the form of first-order logic expressions.

Employing more formalism removes ambiguity, but it does not automatically increase the comprehensibility of the specifications themselves. Hall [13] states that there are three ways to make formal specifications more comprehensible.

- Paraphrase the specification in natural language.
- Demonstrate consequences of the specification.
- Animate the specification.

Animating the specification in the context of defining views may be infeasible due to the fact that the formal specifications in this case are mostly definitions of constraints, which is difficult to visualize. However, one can boost the understandability of individual view specifications by using consistent graphical notations.

6. Concluding remarks and further research

The main contribution of this paper is the proposal of a framework for creating refinement-based taxonomies of views. We established the concept of the baseline view and defined a rigorous model of the baseline view, which is

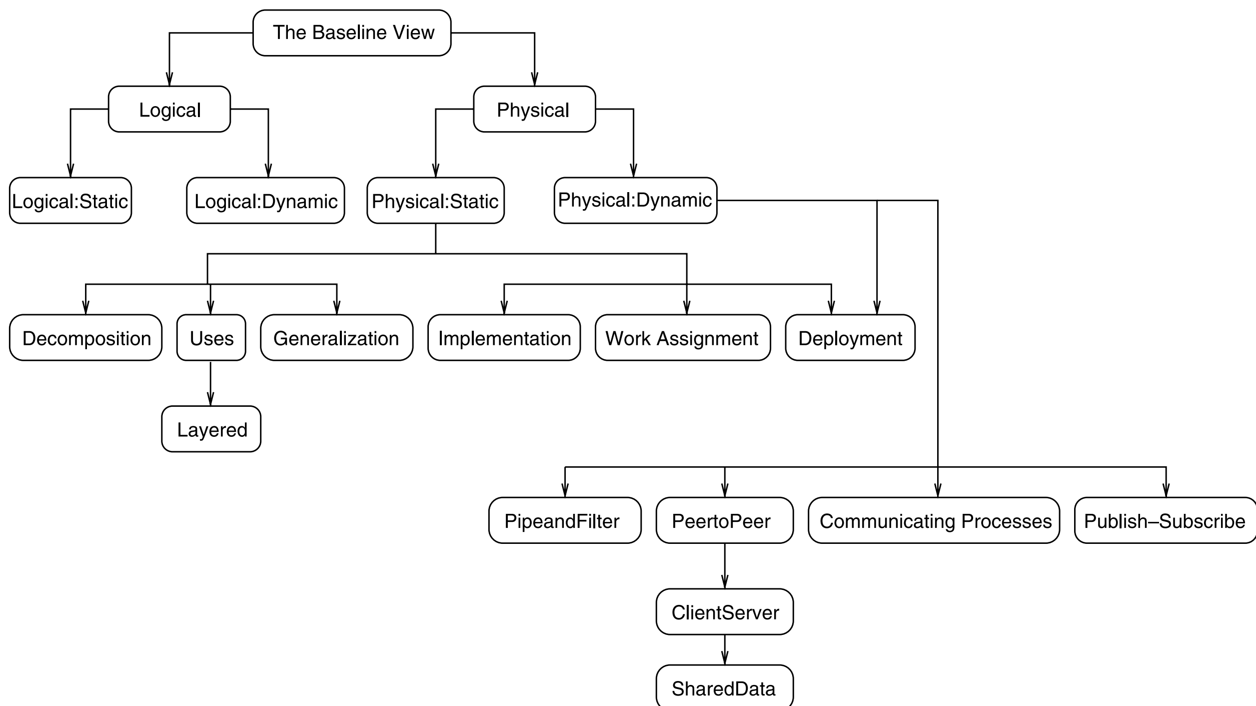

Fig. 11. The final taxonomy.

a fundamental conceptual basis in developing derived views through refinement.

The highlights of our approach include:

- mechanisms to unambiguously define a view in a repeatable manner,
- a novel theory to systematically create an unlimited set of new views and establish seamless relationships between the new views and an existing taxonomy, and
- an ability to check the validity of a view by inspecting whether a view has any discrepancies against its definition.

Currently, we are investigating ways to extend our meta-taxonomy into a full-blown view description language. The essential building blocks (the baseline view constructs/constraints and an extension mechanism) for specifying a view are distilled to our meta-taxonomy framework and provide a good starting point to build the metamodel of such a language.

One of the foreseeable advantages of a language dedicated to describing architectural views is its ability to automate the process of validating an instance of a view against its original definition. For flexibility, our framework does not designate a uniform formalism to describe the additional constraints required to define a derived view. We have also limited the scope of our paper so that we do not sidetrack our discussion into the problem of developing standardized canonical or graphical notations. However, standardization in notations and constraint specification is mandatory to achieve the automatic validation of an instance of a view, and the view description language will be an indispensable tool for standardization.

The view description language will also facilitate the adoption of our meta-taxonomy framework and the exchange of view specifications between different organizations.

References

[1] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology 6 (3) (1997) 213–249.

[2] R.J. Allen, A Formal Approach to Software Architecture. PhD Thesis, School of Computer Science, Carnegie Mellon University, 1997.

[3] R.J. Allen, D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology 6 (3) (1997) 213–249.

[4] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, Reading, MA, USA, 1998. ISBN 0-201-19930-0.

[5] B. Boehm, A spiral model for software development and enhancement, Computer 21 (5) (1988) 61–72.

[6] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, Extensible Markup Language (XML) 1.0, second ed. W3C Recommendation, October, 2000.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Documenting Software Architectures, Addison Wesley, Reading, MA, USA, 2003. ISBN 0-201-70372-6.

[8] P. Clements, R. Kazman, M. Klein, Evaluating Software Architectures, Addison-Wesley, Reading, MA, USA, 2002. ISBN 0-201-70482-X.

[9] E.M. Dashofy, A. van der Hoek, R.N. Taylor, A highly-extensible, XML-based architecture description language, in: Proceedings of the Working Conference on Software Architectures, Amsterdam, Netherlands, IEEE Computer Society and IFIP, 2001, pp. 103-112.

[10] E.M. Dashofy, A. van der Hoek, R.N. Taylor, An infrastructure for the rapid development of XML-based architecture description languages, in: Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, IEEE Computer Society, 2002.

[11] D. Garlan, A. Kompanek, An Activity Language for the ADL Toolkit Working Draft, August, 2000.

[12] D. Garlan, R.T. Monroe, D. Wile, ACME: an architecture description interchange language, in: Proceedings of CASCON (Center for Advanced Studies Conference) '97, Toronto, Ontario, Canada, Center for Advanced Studies & National Research Council of Canada, 1997, pp. 169–183.

[13] A. Hall, Seven myths of formal methods, IEEE Software 7 (5) (1990) 11–19.

[14] P. Herzum, O. Sims, Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise, Wiley, New York, 1999. ISBN: 0471327603.

[15] C.A.R. Hoare, Communicating Sequential Processes, Prentice/Hall International, Englewood Cliffs, NJ, USA, 1985. ISBN 0-131-53271-5.

[16] C. Hofmeister, R. Nord, D. Soni, Applied Software Architecture, Addison-Wesley Professional, Reading, MA, USA, 1999. ISBN 0-201-32571-3.

[17] IEEE, IEEE recommended practice for architecture description, IEEE Standard 1471 (2000).

[18] A. Jaaksi, J. Aalto, A. Aalto, K. Vättö, Tried and True Object Development: Industry-Proven Approaches with UML (SIGS: Managing Object Technology), Cambridge University Press, Cambridge, 1998. ISBN: 0-521-64530-1.

[19] J. Jacky, The Way of Z, Cambridge University Press, Cambridge, 1997. ISBN 0-521-55976-6.

[20] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, Object-Oriented Software Engineering: A Use Case-Driven Approach, Addison-Wesley, Reading, MA, USA, 1992. ISBN 0-201-54435-0.

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

[22] P. Kruchten, The Rational Unified Process, An Introduction, second ed., Addison-Wesley, Reading, MA, USA, 2000. ISBN 0-201-70710-1.

[23] P. Kruchten, The Rational Unified Process: An Introduction, third ed., Addison-Wesley, Reading, MA, USA, 2003. ISBN 0-321-19770-4.

[24] N. Medvidovic, P. Oreizy, R.N. Taylor. Reuse of off-the-shelf components in C2-style architectures, in: Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, IEEE Computer Society, 1997, pp. 692–700.

[25] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, J.E. Robbins, Modeling software architectures in the unified modeling language, ACM Transactions on Software Engineering and Methodology 11 (1) (2002) 2–57.

[26] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70–93.

[27] Object Management Group, OMG unified modeling language specification, Version 1.5, March 2003.

[28] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52.

[29] A. Purhonen, E. Niemelä, M. Matinlassi, Viewpoints of dsp software and service architectures, Journal of Systems and Software 69 (1–2) (2004) 57–73.

[30] E. Roman, S. Ambler, T. Jewell, F. Marinescu, Mastering Enterprise JavaBeans, 2nd edition, Wiley, 2001. ISBN 0-471-41711-4.

[31] W. Royce, Managing the development of large software systems: concepts and techniques, in: Proceedings of WESCON, Los Alamitos, CA, USA, IEEE, IEEE Computer Society, August 1970, pp. 1–9.

[32] M. Shaw, D. Garlan, Software Architecture: Perspective on an Emerging Discipline, Prentice-Hall, Englewood Cliffs, NJ, USA, 1996. ISBN 0-131-82957-2.

[33] D. Soni, R. Nord, C. Hofmeister, Software architecture in industrial applications, in: Proceedings of the 17th International Conference on Software Engineering, Seattle, Washington, USA, IEEE Computer Society, 1995.

[34] Sun Microsystems, Inc., Java™2 platform enterprise edition specification, v1.4, http://java.sun.com/j2ee/1.4/ download.html#platformspec, November 2003.