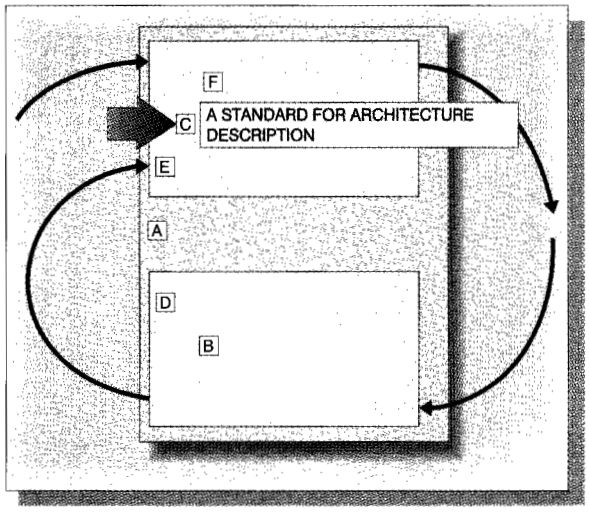
A standard for architecture description

by R. Youngs
D. Redmond-Pyle
P. Spaas
E. Kahan



A profitable information technology (IT) services organization is dependent on widespread asset harvesting (from previous engagements) and scalable asset deployment (into current and future engagements). This activity demands consistency of terminology and notation in the creation and use of engagement artifacts, including work products. This paper presents a standard for architecture description in which a set of conventions for terminology and notation is used to describe and to express the organization of the architecture for an IT system. This standard, the Architecture Description Standard (ADS), is intended to be used by the IBM architecture community. The emphasis is on a minimal set of shared concepts that can be effectively taught to a broad range of IT architects with different skills and that is usable in practice.

This paper focuses on the conventions, terminology, and notation that are needed to support the harvesting and reuse of reference architectures. (Within the context of papers in this issue, it would be helpful to readers if this paper is read before the papers on reference architectures and engagement experiences that are included in this issue.)

The business background

In projects that are developing computing systems for business solutions, it is generally recognized that the use of predefined, reusable assets in the form of architectural, analysis, and design patterns can enable large reductions in project cost, time scale, and risk. However, effective large-scale deployment of architectural patterns is dependent on key concepts, terms, and notations being used consistently, and being understood and accepted across a broad community of information technology (IT) architects and systems integrators. Without a common language, deployment is likely to be patchy, inefficient, and error-prone and to require huge support resources. Lack of consistency seriously inhibits scalability.

In 1996 and 1997, IBM's Global Industries business unit, which has the mission of developing and supporting packaged industry solutions, recognized the need to adopt an improved, asset-based approach

0018-8670/99/$5.00 © 1999 IBM

to its product development. At the same time, IBM's Global Services business unit had independently concluded that IBM and its customers, and IBM's services professionals worldwide, would benefit greatly from an asset-based approach to solutions development in which architectural and design assets would be gathered from completed projects and redeployed on many other, similar projects. Both business units agreed that the increased emphasis on assets and work products for development and for services engagements necessitated a comprehensive metamodel that would underpin the description of those assets and work products more precisely and effectively.

Independently of the Architecture Description Standard (ADS) project, the Enterprise Solutions Structure (ESS) project had already developed a specific metamodel used to document the (mostly technical) frameworks that it was developing. This metamodel, in a simplified form, was implemented in a Lotus Notes**-based tool, which was used to distribute these assets to users of ESS.

The Architecture Description Standard project was created to develop a more wide-ranging version of this metamodel and the semantic descriptions to support it. The output from this project, ADS, provides a common language through the definition of a formal metamodel, a glossary (see the Appendix), and a detailed semantic specification.

The primary audience for the standard consists of IT architects working on solution development and deployment projects. Such work might be either in the context of a client engagement or a development project within an IBM solution development organization. In the former context, assets in the form of work products conforming to the standard may be selected, customized, and used to build IT systems for the customer. In the latter context, developers will create work products conforming to the standard which can then be widely deployed.

Such work products will typically contain descriptions of groups of entities from the metamodel, documented in the form prescribed by ADS. Thus, both providers and consumers of work products will benefit from a common, unambiguous definition. Within a single project, ADS will enable more precise, unambiguous, and semantically rich communication among project personnel.

The standard is intended to be used for solution development and deployment across the IBM Corporation worldwide. It is the foundation for the Systems Integration/Application Development (SI/AD) method and its associated work products. It will be initially deployed to all SI/AD architects via SI/AD education classes. It will also be deployed to a wider range of architects via classes that are currently under development. It is also the foundation for the structure and terminology of asset libraries (for example, ESS).

This paper summarizes the main concepts in ADS. A full description is available in IBM's formal ADS documentation.

Influences on ADS technical strategy

Several themes contributed to the technical strategy adopted for the Architecture Description Standard.

Requirements for an ADL. As discussed previously, there is a business requirement to be able to express architectural work products and assets in a consistent, unambiguous form. ADS is a *language* for describing and communicating architectural concepts. The phrase *architecture description language* (ADL) is commonly used for this type of standard, and there are an increasing number of ADLs available that broadly address the same concerns as ADS—they are all formal languages that can be used to describe the architecture of an IT system.[1] They differ from modeling or programming languages in that their focus is mainly on architectural concepts—abstractions of components, connections, protocols, and the behavior of the complete system.

Because of some of IBM's unique requirements (specifically, those relating to developing asset-based services with a multiskilled work force), no existing ADL was identified that met all requirements and was deployable to IBM's architecture community.

Integration of application development with infrastructure design. One of the conclusions from earlier work was that the success of major solution development projects in IBM often depended critically on the integration of application and infrastructure.[2] Therefore, one of the principles that guided the development of ADS was the recognition that infrastructure design is a specialized skill and that its exponents habitually deal with concepts, entities, and methods that are different from those in "traditional" application development. This recognition led the ADS project to divide the architectural model into

two parts: a functional aspect and an operational aspect.

The operational model, described below, is focused specifically on aspects of architecture necessary for the infrastructure designer to perform his or her job. Although ADS defines these two models, in actual fact they share many entities and can be considered as dealing with the same material, but from different perspectives. Hence, we use the terms "functional aspect" and "operational aspect," and it is through the formal definition of the way entities are shared and used across these aspects that ADS contributes to the integration.

Precision and consistency. Although it might seem desirable to strive to achieve the highest levels of precision and consistency for a technical standard such as this one, we do not believe that it is necessarily the case. To assist the rapid development of asset-based services in IBM, the emphasis in the ADS project was to provide a practical standard that can be readily adopted by a wide range of people in development projects.

One objective, therefore, was to define an architecture description standard sufficient to enable consistent definition and use of architectural templates by practitioners and to underpin the architectural aspects of IBM methods (for example, the SI/AD and IPD, or Integrated Product Development, methods). Another, related, objective was to provide a consistent base of concepts, terms, and notations for the education and training of architects in templates and methods.

Such a standard does not have to be comprehensive to be effective. It only needs to cover the core areas of architecture, which must be defined and understood in a standard way to enable effective deployment of architecture templates. This set, therefore, consists of those concepts that are regularly used in project work products or assets and that need to be standardized for practitioners to do their work.

Although underlying precision and consistency are important (and will be achieved through the metamodel), practicality, trainability, and usability as they apply to practitioners are paramount. The critical factor for success is whether the resulting set of concepts, terms, and notations is small, simple, and accessible enough to be taught to large numbers of practitioners in a broad spectrum of courses.

Exploitation of IBM's existing best practices. In our work developing the Architecture Description Standard, we have attempted to reconcile and synthesize various IBM best practices and combine them with wider industry initiatives. The main IBM sources we have integrated are:

- WSDDM-OT (worldwide solution design and delivery method-object technology)
- WSDDM-ISD (infrastructure design), which itself incorporates the earlier IBM End-to-End Infrastructure Design Method
- ESS, a major asset base of IBM reference architectures and associated architectural assets

Use of industry standards. During the 1990s, the largest and most influential industry initiative on the representation of software systems has been the work undertaken by the Object Management Group[3] (OMG) to produce a Unified Modeling Language (UML). ADS decided to adopt UML (v1.1) as the basis for ADS concepts, terms, and notations because UML is being widely adopted as a *de jure* and *de facto* software modeling standard.[4] It represents a major investment of intellectual effort and conceptual convergence by the world's leading software methodologists. Consequently, most of the major modeling tool vendors will provide CASE (computer-assisted software engineering) tool support for UML.

In addition, many IBM IT architects are already familiar with UML concepts and notations (indeed IBM was closely involved in the development of UML), and most of the key concepts in WSDDM-OT, WSDDM-ISD, and ESS, with which they are familiar, can be mapped to UML concepts.

During the development of ADS, we have identified some limitations in UML for architecture description and areas in which IBM best practice would be lost or diluted if we conformed slavishly to UML. In these areas we have extended UML so that valuable concepts or notations are not lost. This extension has proved to be most necessary in the area of infrastructure design, where the UML coverage of important concepts such as connections between nodes, and service-level requirements, is limited.

The technical strategy has therefore been to adopt a pragmatic approach: integrating concepts from IBM's WSDDM-OT, WSDDM-ISD, and ESS; expressing these concepts in UML terms and notation wherever possible; extending UML where required while ensuring that the concepts are usable by practitioners.

The authors believe that this strategy has achieved a coherent and usable architecture description standard that carries forward the strengths of the methods and standards from which it is derived.

The architecture definition standard

This section begins with a definition of architecture and then describes its various characteristics.

What is an architecture? The IT industry has proposed numerous definitions for the concept of architecture, widely varying in scope and emphasis. The main focus of ADS is on describing the structure of "components," the relationships between them, and the way in which they interact dynamically. Hence, ADS has adopted the following definition:

> The architecture of an IT system is the structure or structures of the system, which comprise software and hardware components, the externally visible properties of those components, and the relationships among them. (Adapted from Bass et al.[1])

Note that this definition incorporates hardware components in the scope of a systems architecture. In practice, most project activity is concerned with software architecture, design, and implementation. Therefore, it might be argued that a purely software scope is sufficient—ADS would then describe a software architecture. Occasionally, however, it is necessary to include hardware components in an architectural definition (usually where specialized or unfamiliar hardware devices are needed). Also, through its operational aspect, ADS brings a closer focus on describing computer platforms and their physical connections and on delivering service levels. Therefore, it is appropriate that ADS should include hardware.

Functional and operational aspects. In the context of a development project, a complete IT system architecture serves multiple purposes, among them:

- Breaking down the complexity of the IT system so that developers can analyze and design components that are relatively isolated from one another
- Analyzing the functionality so that required technical components (or infrastructure) can be identified
- Assisting in the analysis of service-level requirements so that the means of delivering them can be designed
- Providing a basis for the specification of the physical computer systems on which the IT system will execute and the mapping of components onto these computer systems

In large projects, a division of responsibilities becomes necessary for the simple reason that a single person cannot possibly be skilled in all the technologies, methods, tools, and techniques needed for all these purposes. Also, the activities of any large project need to be partitioned so that small groups (subprojects or work groups) can manage their own creative work, with the integration of the whole project being performed at a higher level. In practice, large projects include work groups concerned primarily with application design and development. Their focus is on the first of the above purposes. Other groups are concerned with infrastructure design and development, and they focus on the last three purposes. Each group has specialized techniques to address its particular concerns.

The Architecture Description Standard reflects this separation of concerns by identifying two main aspects of architecture: the functional and the operational aspect.

The focus of the functional aspect is on describing the *function* of the IT system and is primarily concerned with the structure and modularity of the software components (both application and technical), the interactions between them, their interfaces, and their dynamic behavior (expressed as collaborations between components).

The focus of the operational aspect is on describing the *operation* of the IT system and is primarily concerned with representing network organization (hardware platforms, connections, locations, topology, etc.), where software and data components are "placed" on this network, how service-level requirements (performance, availability, security, etc.) can be satisfied, and the management and operation of the whole system (capacity planning, software distribution, backup, and recovery).

The aspects are summarized in Table 1.

Functional aspect concepts. The concepts and modeling notations used to describe the functional aspect of an IT system are discussed in the following subsections.

Table 1 Aspects of architecture

Functional Aspect	Operational Aspect
Described in: component models	Described in: operational models
Key concepts: components, interactions, collaborations, interfaces, operation signatures	Key concepts: nodes, connections, locations, network topology, service-level characteristics
Key activities: structuring, providing functionality	Key activities: placement, delivering service levels

Components and relationship diagrams. The functional aspect is represented in terms of components and the relationships between components.

A component is a modular unit of software[5] functionality, accessed through one or more interfaces. The functionality and state of the component are only externally accessible by using these interfaces (that is, they are encapsulated). Component is the primary concept used for modular design, with the well-established design principles of information hiding and of seeking high cohesion and low coupling.

The primary notation used for components in ADS follows the UML class diagram notation. (We represent a component by the UML class symbol—a rectangle, optionally with a section for operations.) Relationships between components are shown in a component relationship diagram in which we show a usage relationship between two components if one component uses the interface(s) of another component. This usage can be thought of as being based on a contract between the components; that is, there is an agreement between the two components about the services they can request from each other. We use the UML association notation to show this. Showing such a usage relationship implies that the components can communicate with one another. The design of the infrastructure must ensure that the necessary physical connectivity can occur.

Figure 1 shows an example of a component relationship diagram for a workflow system.

The component relationship diagram notation has a line with an arrowhead pointing from the "using" component to the "used" component. For example, in Figure 1 the workflow client application uses operations provided by the workflow engine component. Where each component uses (and therefore depends on) the other, the line can have arrows at each end. For example, the workflow engine uses the external workflow engine, and vice versa. (This interpretation slightly extends the standard UML arrowhead notation meaning direction of association navigation.)

Although the line could be read as a data flow or a message, this interpretation is not correct. A usage relationship between two components typically relies on one or more interfaces with many operations (and corresponding message types). Some of these messages may be passed in the opposite direction from the arrow (for example, callback), and data commonly flow in both directions.

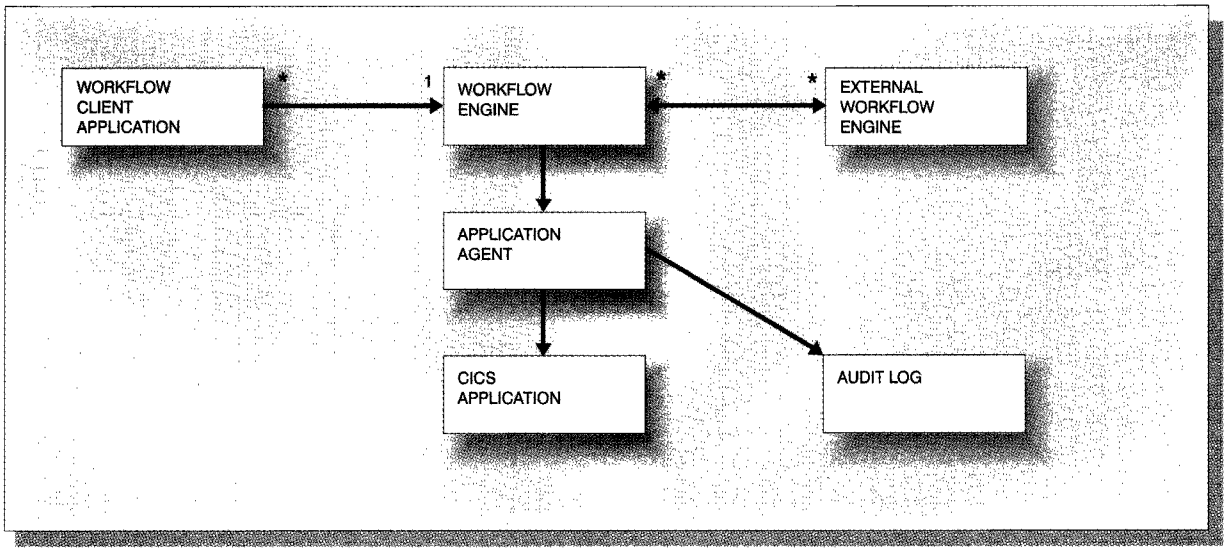
The component relationship diagram can optionally show the multiplicities of relationships. For example, Figure 1 shows that each workflow client application interacts with a single workflow engine and that the workflow engine interacts with multiple clients (as shown by the 1 and * at the ends of the connecting arrow).

A component relationship diagram is essentially a static specification of the usage between components. It shows how components can use one another, rather than actual message flow or dynamic behavior (for these latter items, see the component interaction diagram shown later in Figure 3).

Composition of components. Complex components are frequently composed from simpler components. This arrangement can be visualized by showing one component inside another.

Figure 2 shows the component relationship diagram with the audit log encapsulated within the application agent. When one component is composed of another, the services of the contained component are not accessible from outside the composite component; that is, they are encapsulated. For example, in Figure 2, the application agent uses the audit log, but no other users or components can use the audit

Figure 1 Component relationship diagram



log. Encapsulation does not imply that the audit log is physically contained in the application agent, but that the application agent has a private relationship with it.

It would not be appropriate to show the Customer Information Control System (CICS*) application encapsulated in Figure 2, as its external interfaces may be used by multiple components.

(This is a standard UML class diagram notation for composition. If available tools do not support it, an alternative UML notation is the diamond symbol for aggregation.)

Describing component behaviors. We describe the required externally observable behavior of an IT system in terms of the widely used concepts of use cases and use case scenarios. For example, the statement "Withdraw Funds from ATM" is a use case, and "Successfully withdraw $100 from account 12345 at midnight" is a scenario of this use case (that is, one particular path through the use case).

In creating the architecture of a system, we need to allocate the responsibilities of the system as a whole to the components of the system. For each scenario, we divide up the system responsibilities among a number of components (e.g., one component might run the user interface, a second might manage workflow, a third might store data). To model how the components work together to process a scenario at run time, we have the concept of a collaboration.

Figure 2 Component relationship diagram showing component composition

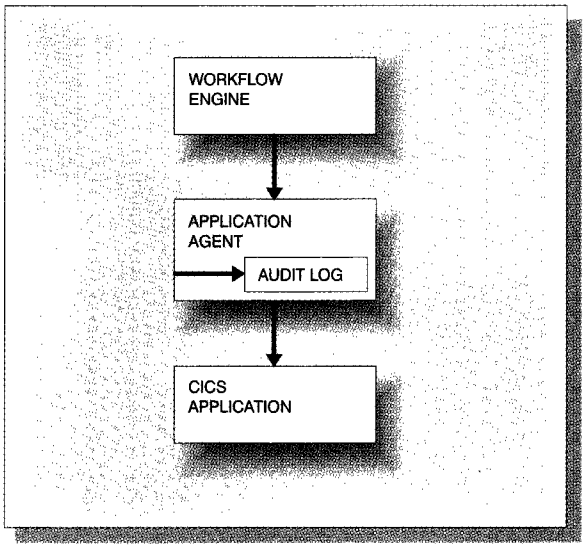
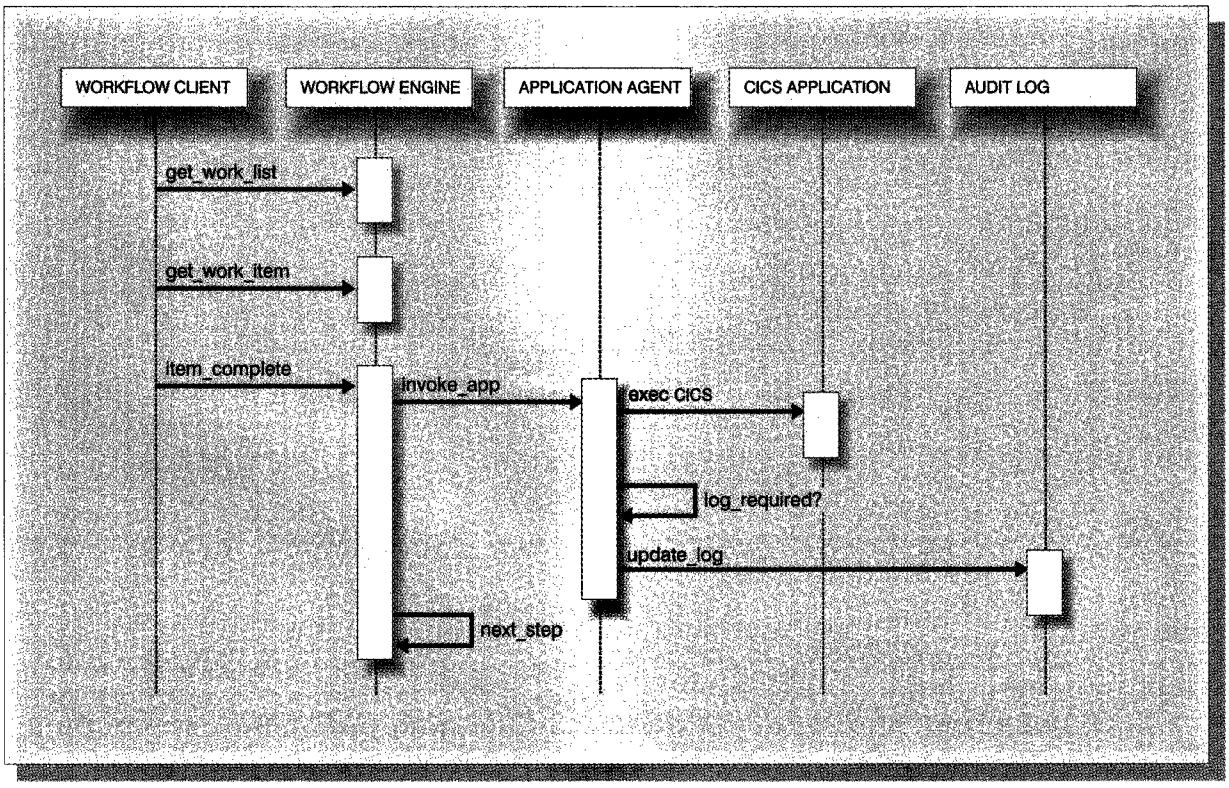


Figure 3 Component interaction diagram



A collaboration (among components) is a sequence of operations, identifying which components perform operations and which request operations, reflecting the time sequence. Collaborations are the primary way of modeling the dynamic behavior of components.

Collaborations are visually represented using component interaction diagrams (using standard UML notation). An interaction diagram shows the messages exchanged between components during a single collaboration; that is, it is an execution trace.

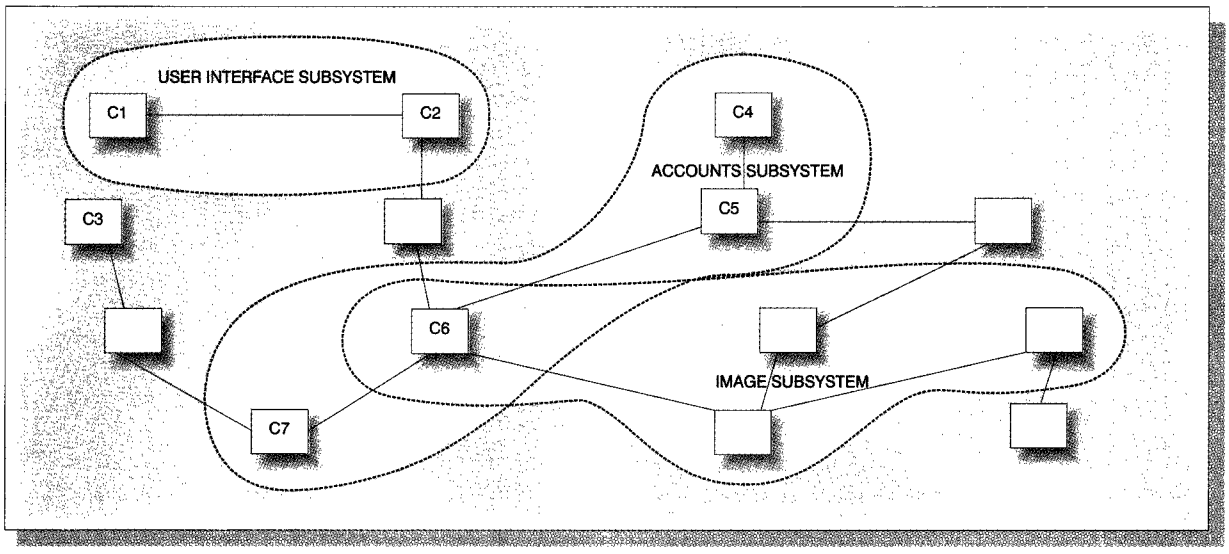
Figure 3 shows a component interaction diagram for the workflow system above. Each vertical line (column) represents a component that participates in the collaboration (e.g., workflow client, workflow engine). Each horizontal line represents a request, in the direction of the arrow, from one component to another. The name of the request (= invoked operation) is shown on the line (e.g., get_work_list). The names, types, and values of parameters can also be shown if required (but are omitted in Figure 3).

We use the standard UML sequence diagram notation to model component collaboration. UML also defines a semantically equivalent notation, the collaboration diagram, which can be used if required. (The collaboration diagram format shows the component topology more clearly but tends to make the time sequence of messages harder to see.)

Structuring of components. One of the most important architectural design processes consists of structuring the system as a whole into a suitable structure of components and relationships between components. This structuring addresses a number of concerns, including:

- Allocation of responsibilities to components, in such a way that each component has a cohesive set of responsibilities and redundancy is avoided
- Partitioning of components to take into account distribution requirements (For example, an initial component may have to be split into a client and a server component.)

Figure 4 Component diagram showing subsystems as collections of components



- Optimizing the component structure to satisfy service-level requirements such as performance
- Incorporation of reusable components in the design, accommodation of legacy systems, or other constraints

The notations illustrated in Figures 1 to 3 are used to represent possible component structures and collaborations during this structuring process. It is common for the structuring to evolve through several elaboration points.

Subsystems. When describing an IT system, we often need to talk about some part of the system. Subsystems may be defined and used for several purposes, for example, to organize a large IT system into smaller "chunks" and to make it more comprehensible or manageable (e.g., accounts subsystem, inventory subsystem, system management subsystem) so that it can be more easily described to other people. It is also necessary to divide the system so that work can be allocated to development teams (e.g., one subsystem per team).

A subsystem is a subset of the components in an IT system. It can be defined by drawing an irregular line around some of the components (see Figure 4). For example, in Figure 4, the accounts subsystem consists of components C4, C5, C6, and C7.

Subsystems may overlap, e.g., component C6 is part of the accounts subsystem and also part of the image subsystem.

A subsystem may span platforms, i.e., the components in a single subsystem can be placed on several platforms. For example, a data access subsystem may have a database server component, a data access gateway, and a client data access port, each running on a separate platform (similarly for an Object Request Broker [ORB], a workflow subsystem, an accounts subsystem, or a systems management subsystem).

Domains and architectural templates. In addition to describing actual architectural components, it is important to be able to describe the way in which they interact and collaborate. The Architecture Description Standard Semantic Specification defines the concept of a domain, which is a set of structural and behavioral patterns that describe some part of the architecture of an IT system (e.g., workflow, transactionality, user interface, network communications). The description of a domain is expressed as a set of collaborations between components and the interfaces (defining component roles) used by these collaborations.

One of the prime motivations for domains is reuse—domains can represent architectural patterns (or

templates)[6] that occur across many components and may be observed (and therefore reused) in many IT systems. Since better reuse of assets is an expected benefit of ADS, the alternative terminology of "architectural template" is used in this paper and elsewhere. This emphasizes the reuse aspect and is consistent with the definition of the architectural template work product type, described in the IBM SI/AD method.

There are several well-known examples of architectural templates, often described by related terms such as reference model, reference architecture, domain description, or even industry standard.

Widely known examples of architectural templates include:

- The Smalltalk Model-View-Controller pattern: A standard set of component roles (model, view, and controller) for user interface management, the interfaces each role must support, and the collaborations which ensure that the user interface (view) is coordinated with the model.
- The Web architecture: A template in which the browser and server collaborate in well-understood ways (defined by HTTP—HyperText Transfer Protocol) to present information across the Internet.
- The Workflow Reference Model from the Workflow Management Coalition:[7] A set of patterns of collaboration between standard component roles (invoking application, workflow enactment service, invoked application, monitoring application) and a detailed workflow application programming interface supported by each role for a component to be "workflow-aware."
- The OMG Object Transaction Services (OTS) specification: A standard set of component roles (transaction manager, resource manager, resource) and required interfaces for transaction management in a distributed object environment.
- The International Organization for Standardization-Open Systems Interconnection (ISO-OSI) seven-layer model: An architectural template for network communications with well-defined layer responsibilities and interfaces.

The same diagrams used for describing component structure and component interactions are also used in architectural templates.

Operational aspect concepts. This section introduces the concepts and notations used in the operational model, which describes the operational aspect of IT system architecture.

Network diagram. The network diagram represents network topology and shows where software components are placed on the nodes in the network (Figure 5).

The central concept is *node*. A node is a platform on which software executes. During early stages of the design process, a node represents a potential platform before decisions have been made about how to map it to actual platforms.

The network diagram in Figure 5 shows nodes as rectangles (ideally, where diagramming tools permit, as UML cuboids). Each node has a name and (optionally) the number of instances (in parentheses). Connections represent physical data paths between nodes (e.g., by local area network—LAN, wide area network—WAN, dial-up, wireless) and show the shape of the network.

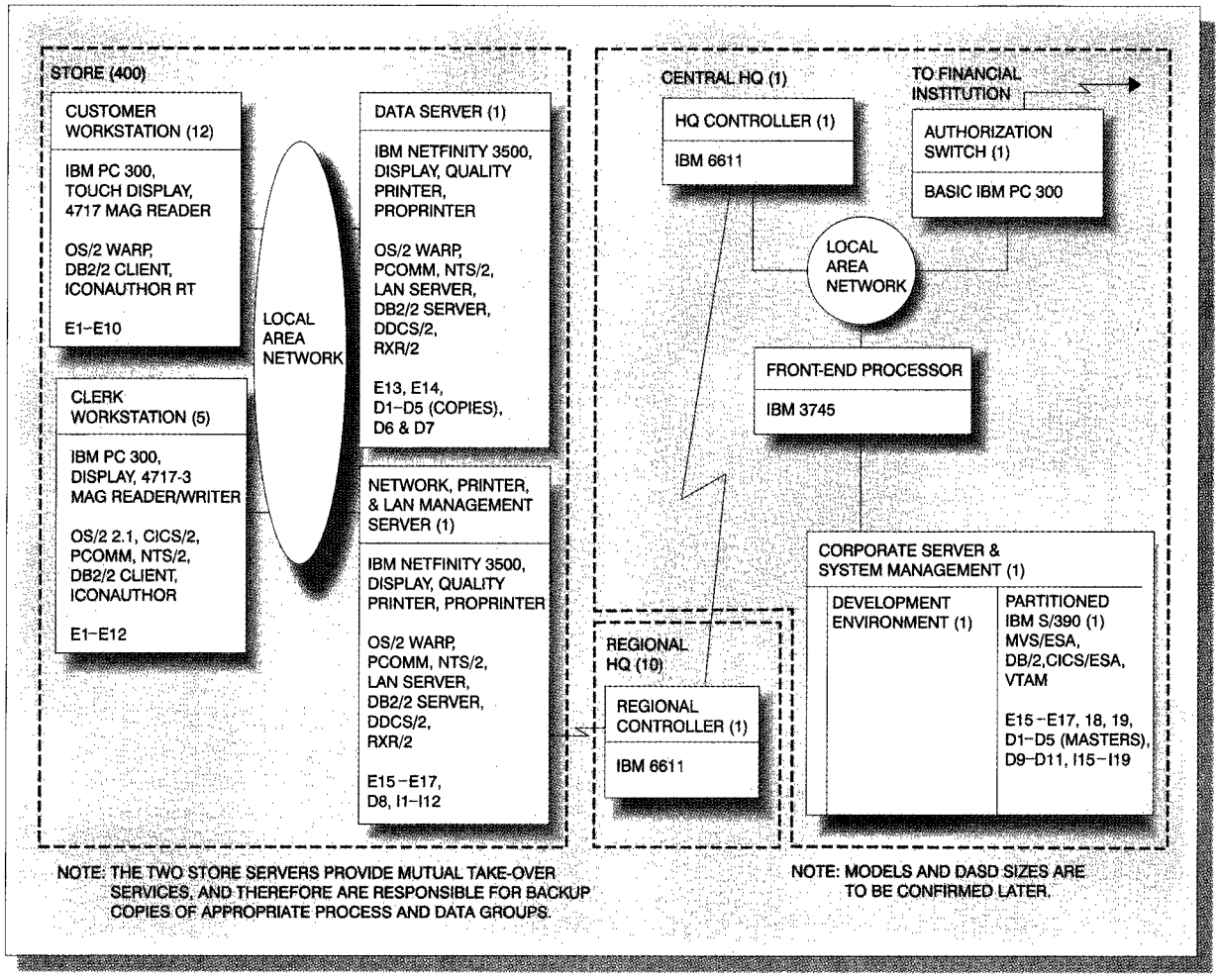
Deployment units are placed on the nodes. A deployment unit is the smallest unit of software or data about which an architect makes a placement decision. Deployment units are shown as named items on each node. In practice, only the most significant units are shown in this diagram; otherwise it becomes too cluttered.

A deployment unit consists of one or more components. In practice, we often need to regard the data aspect (or state) of a component as being in one deployment unit and the execution of a component as being in another deployment unit. Sometimes we also need to distinguish the installation aspect of a component as being in a third deployment unit. Deployment units may be named to reflect this need (e.g., in Figure 5, E1–E14 are execution deployment units, D8 is a data deployment unit) or they may be annotated (e.g., component name [E], CName [D, I]).

Nodes are grouped together in locations. A location is a geographical entity (e.g., a zone or building type) and is shown by a dashed line around one or more nodes. Several network diagrams may be used to represent different aspects of a single IT system (e.g., operational system, systems management, developmental environment).

The network diagram is based on the UML deployment diagram notation with some extensions. Nodes correspond to the UML concept of node.

Figure 5 Network diagram



Walkthrough. One of the main ways of confirming that the operational aspect of a system is feasible and acceptable is to perform end-to-end walkthroughs of collaborations. A walkthrough is a narrative description of how the IT system processes a use case scenario, tracing the collaboration around the system. It is used to assess the operational behavior of the system, specifically whether the system will be able to satisfy service-level requirements such as security and availability.

Walkthroughs are usually documented in text (as a sequence of paragraphs) and cross-referenced to scenarios or collaborations. They can be documented using component sequence diagrams if required.

Placement. A key concern in the design of the operational model is placement—deciding how to group components into deployment units and on which nodes to place them. Placement is influenced mainly by what data the users are operating on, together with where the system users are and what activities the users perform. It is also influenced by nonfunctional requirements (see next subsection).

When implementing a commercial software package, the deployment units to be placed are predefined. In contrast, when a solution is being designed and developed, the placement process determines the grouping of components into deployment units and influences the partitioning of components.

Nonfunctional requirements. The other key influences on placement are the nonfunctional requirements, for example:

- Performance (end-to-end response time for specified services)
- Availability (e.g., 8.00–20.00 weekdays, 24 × 7, etc.)
- Security policies, etc.

A nonfunctional requirement is a quality requirement or constraint that an IT system must satisfy.

Service-level requirements (such as those listed above) are an important type of nonfunctional requirement. Nonfunctional requirements also include system qualities such as maintainability, even though these relate to attributes of the development process, rather than to attributes of the operational system (like service levels).

All kinds of constraints on an IT system are also represented using this construct, including business constraints (e.g., geography of locations), IT standards (e.g., CORBA**—Common Object Request Broker Architecture—compliance), and current infrastructure constraints (must run on specified existing middleware).

In ADS, nonfunctional requirements may be attached to any component or node, and the operational model is developed by placing components onto nodes according to their nonfunctional requirements and then successively refining placement decisions. Although nodes themselves should strictly be modeled as sets of (hardware) components, it is normally sufficient to regard the node hardware and operating system as a simple container for software components, with the interfaces to the hardware components being provided by the operating system.

Integration of functional and operational aspects. As noted earlier, one of the major challenges in developing systems is achieving more effective integration between what has traditionally been application development and infrastructure design areas of work. ADS separates the concerns of the functional aspect (collaborating components) and the operational aspect (satisfying service requirements). At times it is important to be able to work on one aspect without considering the other.

ADS facilitates the integration of functional and operational aspects in five main ways:

A. Component placement—Components (functional) are placed on nodes (operational) in order to meet the service-level requirements and other quality requirements of the IT system such as manageability. Co-located components are grouped into deployment units to ease placement. Where required, the stored data of the component can be placed on a separate node from where the data execute. Interactions (functional) are mapped to connections (operational).

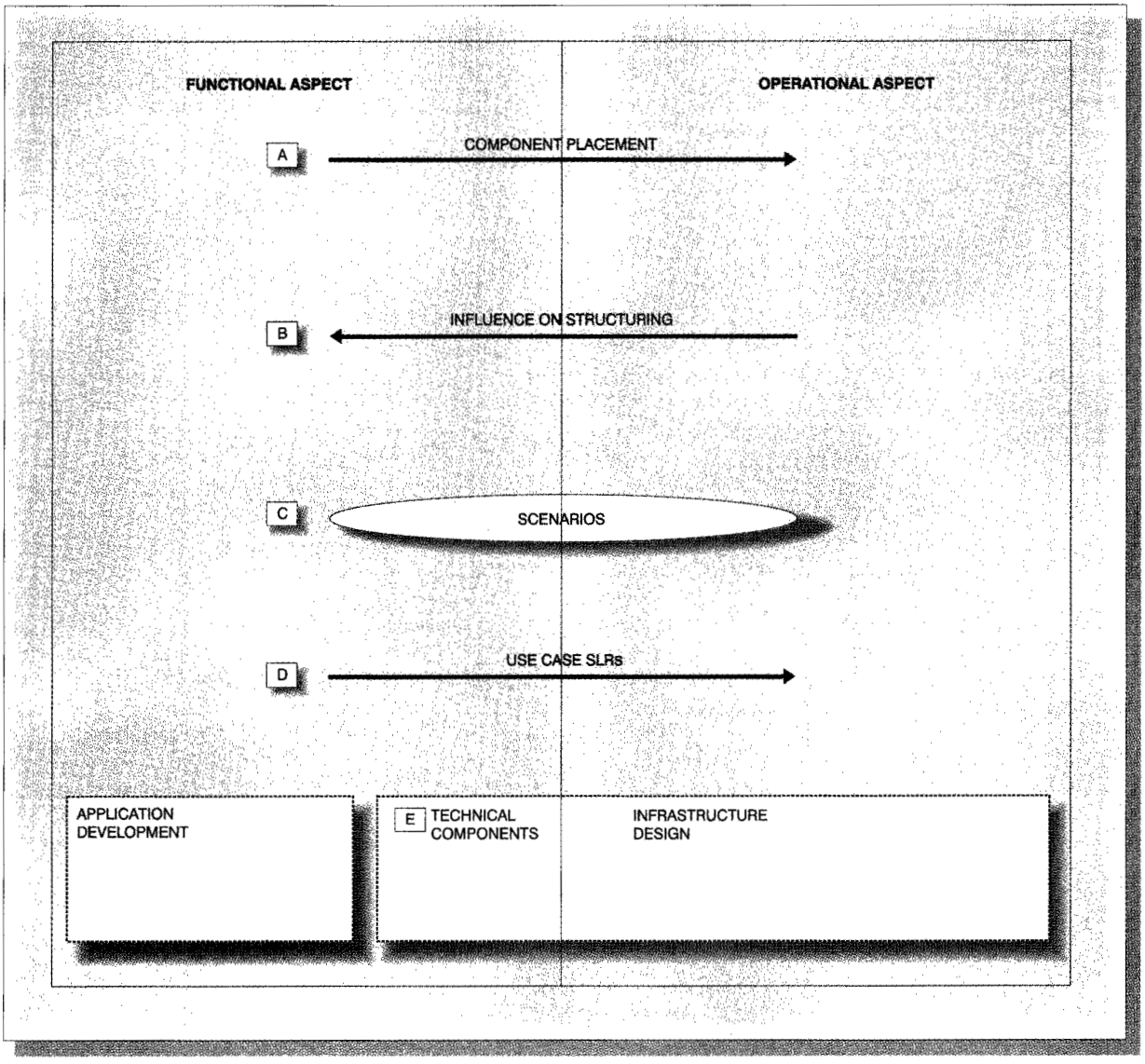
B. Component structuring—Operational concerns influence component structuring. Operational issues are frequently neglected during application development. ADS shows the concepts and issues in the operational model that have a direct effect on the component model. Components are (re)structured to take into account distribution requirements, operational constraints, and the need to achieve service-level requirements. For example, a component may be split into a client component and a server-based component, with a usage relationship between them, to achieve a response time objective on the client.

C. Scenarios—ADS provides use case scenarios as a unifying theme that runs through both functional and operational models. A use case scenario is realized in the functional model as a collaboration between components, with a sequence of operations executed. This collaboration (based on the same scenario) can be represented as a walkthrough in the operational model to validate that the placed components can achieve required service levels.

D. Use case service-level requirements—ADS enables effective integration of service-level requirements (SLRs) between functional and operational aspects. ADS recommends the specification of SLRs for use cases, which are then carried over to their associated collaborations as component collaborations. SLRs in this form are a valuable input to validate that the operational model will satisfy requirements. However, SLRs are also attached to many other entities, including the IT system itself.

E. Technical components—Last but not least, ADS facilitates integration by treating the technical components as part of the functional aspect. Technical components are components represented in a component model with component sequence diagrams in exactly the same way as ap-

Figure 6 Integration of functional and operational aspects



plication components. This treatment enables standard component modeling techniques to be used to represent application or technical integration.

These five forms of integration are illustrated in Figure 6.

Elaboration points. As the development of an IT system proceeds, the nature of the material that developers deal with changes. At the start of a project, abstract concepts such as requirements, use cases, and principles are the primary concern of project personnel. Toward the end of a project, different, more concrete concepts dominate; executable code, hardware specifications, and test cases are some of the major concerns.

However, during the development of an architecture, some entities retain the same basic concept but become progressively more refined and detailed. This progressive refinement takes two forms:

- As more information becomes available, and as the project gains a better understanding of the nature of the problem, more detail is added so that, finally, enough detail is available for a designer or implementor to take over. There is a sense of continuous progression in this refinement.
- At certain points in the process, structural changes have to be made, again reflecting better understanding of the problem, the environment, and the technology available. For example, it may be necessary to divide and merge components to handle nonfunctional requirements such as availability or performance. At this point there is a discontinuity in the development of the structure of the architecture.

Furthermore, in any complex project, the division of responsibilities means that each subproject will need to make its own structural changes, and those changes will need to be synchronized among subprojects. These points, therefore, are obvious places to produce major work products. Where predefined work products are used in a project, it is imperative that both the creator and the user of each work product agree on these key points in the development process.

Elaboration points are places at which major structural changes are made in all relevant parts of the project. (Entities may be split, merged, or regrouped; some may be deemed to be no longer needed, and some new ones may be introduced.) Often, unusually large quantities of information are exchanged between subprojects so that a uniform and coordinated design can be achieved. At the same time, major work products may be exploited or created, and formal reviews of work products may be held. Elaboration points are sometimes equivalent to checkpoints or to the transition between "stages" or "phases" in development methods.

In the current version of ADS, exact elaboration points are not defined.[8] However, the initial and final elaboration points in any project will generally be the same, even though the number and nature of intermediate ones may differ. For example, ADS recommends the following:

- Initial elaboration point: unconstrained by the limitations of technology, geographical distribution, and the customer's environment. At this stage, relatively coarse-grained and undetailed models and specifications are produced.
- Intermediate elaboration point: one, for example, in which the distribution of users and locations is taken into account so that components are geographically distributed, or when new technological components may be identified to support the component distribution.
- Physical elaboration point: fully constrained by the limitations of the technology available, geography, and the environment. Fine-grained and detailed models and specifications are produced. The work products at this point are a complete architectural specification of the IT system, including specifications of hardware and software products so that they may be ordered. This work may include upgrades to installed products or connections. A specification of the operational aspect is produced so that nodes and connections can be configured and components deployed and brought into operation. If any new functionality needs to be created, module specifications are created that can be taken by a developer or programmer to be developed and unit-tested.

Not every aspect need have the same elaboration points. The functional aspect may have an initial elaboration point (usually the first elaboration point encountered in a project), but the operational aspect will usually not. Each aspect may have differing intermediate elaboration points, reflecting the different concerns of each aspect. However, all aspects should have a physical elaboration point.

Readers familiar with the WSDDM infrastructure design (ISD) method will recognize that the end stages of ISD closely correspond to elaboration points.

The concepts are summarized in Figure 7. This figure shows the IT system with its two aspects, functional and operational. Within each aspect, the major concepts are shown, separated by a dashed line.

Notation. ADS defines several standard notations. Wherever possible, these use the OMG UML notations. Table 2 summarizes the notations used in ADS.

Relationship with existing terminology and roles. This section positions some commonly used terminology and roles with ADS concepts. None of the terms in the following subsections is defined in ADS.

Application architecture. An application architecture view emphasizes those components and their interrelationships that provide the application-dependent behavior of the IT system. As such it is a view of a part of the functional aspect of the IT system.

Figure 7 Summary of basic ADS concepts

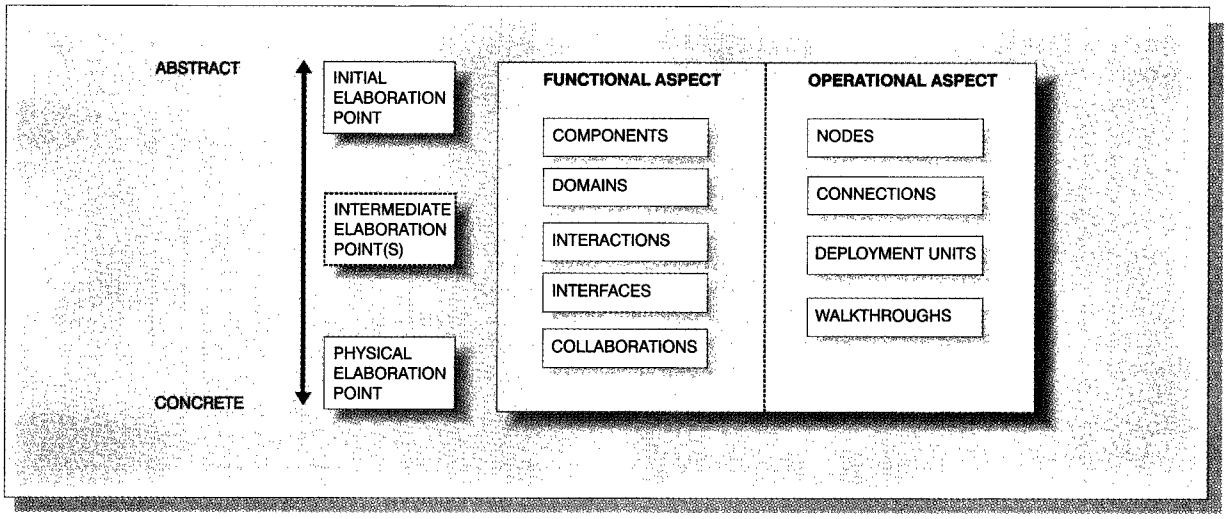


Table 2 Notations in ADS

Notation Name	Where Defined/Comments
Component relationship diagram	UML class diagram, with components in place of classes Any extensions/interpretations defined in a component model Static representation of component usage relationships and composition
Component interaction diagram	UML sequence diagram, with components in place of objects
Component collaboration diagram	UML collaboration diagram, with components in place of objects
Network diagram	Defined in the operational model Based on UML deployment diagram, with extensions

Technical architecture. A technical architecture view emphasizes those components and their interrelationships that provide the application-independent behavior of the IT system. As such it is also a view of a part of the functional aspect of the IT system.

It is important to understand that ADS does not distinguish between a technical architecture and an application architecture. Any such division is somewhat arbitrary, so that no formal distinction is possible. For example, a workflow manager run-time server (a component in ADS terminology) can be thought of as a technical component (part of a workflow manager product), yet it contains business process definitions and information about the business organization—clearly application concepts. This component, therefore, has both application and technical responsibilities.

For this reason, the ADS functional aspect encompasses components present in both application and technical architectures, and uses the same concepts and notations for them.

The application development and infrastructure development roles. Application development methods and practitioners concentrate on the functional aspects of an IT system. Their main concern is, on the one hand, to partition the work in such a way that it can be allocated to development groups and, on the other hand, to build new components in such a way that they support the functional requirements. Service-level requirements (such as availability and performance, etc.) are not the primary concern of the application developer, although they may greatly influence the design of their components (though

other nonfunctional requirements such as maintainability may be a major concern).

The infrastructure designer forms clusters of business and technical components to populate nodes in locations. The formation of these clusters is based on analysis of service-level requirements. This activity results in specifications being drawn up both for nodes and new components. Some of the latter are then passed to the application developers for development, others are acquired from vendors.

This simple approach may ignore the essential feedback that the placement process generates and that should be used to further influence the design of the components. Furthermore, technical components (for example, messaging) may end up in a gray area between the application developers and infrastructure designers, depending on whether they are to be acquired or custom-built. The more rigorous approach and the emphasis on formal, complete component models in ADS will help to eliminate such tendencies.

Enterprise architect role. Enterprise architects (formerly IT architects) build architectural models to describe the functional requirements of the infrastructure based on business objectives, the IT vision, and principles. The enterprise architecture method is a guide for IT architects and consultants involved in defining enterprise-wide architectures for clients. It provides specific guidance on the steps required to develop an enterprise architecture and general guidelines for sizing and estimating, report writing, and quality assurance.

The concepts present in these models reflect the key building blocks of the architecture and the interfaces between them. These building blocks are further refined to represent a reusable set of specifications for which evaluation criteria and standards can be developed, and eventually product selection can be done.

Work is in progress to map enterprise architecture concepts to ADS concepts.

Concluding remarks

The Architecture Description Standard is a key part of the foundation for an asset-based business in IBM. It contains a metamodel of an IT system, a semantic description, and a glossary that are designed to be sufficiently comprehensive for the target user community. It is also designed to be simple enough to be deployed to a user with a wide range of skills.

In this paper we have described how ADS was based on the use of industry standards, particularly UML. We have shown how these standards have been extended in particular areas, such as the operational aspect of architecture, in order to highlight areas that IBM has learned in practice are crucial in the design of large-scale enterprise systems.

We anticipate that through facilitating asset creation and reuse, widespread use of ADS in IBM will deliver considerable benefits, measured in financial terms, as reductions in cost, risk, and time in development projects. Less tangible benefits will be a more concise and precise means of communication between different parts of development teams and between IBM and its clients. In other papers in this issue (on reference architectures and engagement experiences) it can be seen how early versions of ADS have already been successful in helping IBM to structure assets and then reuse them in architecture projects to the benefit of both IBM and its clients.

Acknowledgments

In addition to the authors of this paper, the Architecture Description Standard project team consisted of: John Cameron, Ian Charters, Martin Cooke, and Dave Vanberg (with Ed Kahan as project leader). The following also contributed to the development of ADS, either directly or through its predecessors: John Black, Steve Cook, Paul Fertig, George Galambos, Ralph Hodgson, Deborah Leishman, Tim Lloyd, and John Rothwell.

Appendix: Glossary of ADS terms

The main terms in the Architectural Description Standard are defined below. Entries that are considered to be "core" to ADS are represented in a large-size boldface type. General industry terms, or minor ADS terms, are regarded as "noncore" terms and are shown in regular-size boldface type.

Actor An actor is a human user or external system that interacts with the system being built, by executing use cases. An actor represents a coherent set of roles. Several users can play the same role, and one user can perform several roles.

Examples of actor could include customer service representative or credit authorization service (an external system).

Architecture "The architecture of an IT system is the structure or structures of the system, which comprise software and hardware components, the externally visible properties of those components, and the relationships among them." (Adapted from Bass et al.[1])

Architectural template An architectural template is a set of structural and behavioral patterns that describe some part of the architecture of an IT system (e.g., workflow, transactionality, user interface, or network communications). It describes component roles, the interfaces that components playing these roles must provide, and a set of standard collaborations.

Familiar examples of architectural templates are:

- The Web architecture, in which the browser and server roles collaborate in well-understood ways (defined by HTTP) to present information across the Internet
- The Workflow Reference Model, which defines the standard collaborations and interfaces by which a workflow system can interoperate with both client applications and invoked applications
- The ISO-OSI seven-layer model, which constitutes an architectural template for network communications with well-defined layer responsibilities and interfaces

Architectural templates are useful for promoting standardization and reuse of architectural assets. They can also be used on a project by architects to communicate standard mechanisms to designers.

Collaboration A collaboration is an occurrence of a sequence of operations that realizes a use case scenario. It typically involves collaboration between two or more components.

Collaborations are visualized in collaboration diagrams or sequence diagrams (as defined in UML). As an example, consider the scenario of updating customer details in a client/server system. There is a sequence of operations in which the graphical user interface (GUI) component displays a window, calls a data server component with a request for data, displays the customer details (and amends them), calls the data server to perform an update, etc. This whole pattern of component operations and exchanges between components is a collaboration that "realizes" the scenario. Once components have been placed on nodes, the end-to-end behavior of a collaboration may be assessed and documented using a walkthrough.

Component A component is a modular unit of functionality, accessed through one or more interfaces. A component offers a set of interfaces to the outside world, while encapsulating its own state and behavior. A component should implement a cohesive set of functionality. A component may be composed of other components, which are encapsulated within it.

Components normally represent software (including operating systems) but can also represent firmware (e.g., PC BIOS) or hardware (e.g., encryption device, interactive voice response units).

The functional aspect of an IT system is described by groups of interacting components, ranging from the very large to the very small.

Examples of components include a customer business object, domain name server, an Open Database Connectivity driver (small), database management system (large component), a spreadsheet dynamic link library to be used imbedded in other applications, and a JavaBean**.

Many definitions of component are currently in use. Some of these restrict use of the term component to software implemented in specific technologies such as CORBA, JavaBeans, or ActiveX** that enable self-descriptive interfaces. Our use of component includes these technologies but is not restricted to them.

Connection A connection joins two or more nodes in order to support interactions between components that have been placed on those nodes. A connection may consist of a simple cable between nodes, but it is typically used to represent various kinds of network connections (e.g., LAN, WAN) or communications types (e.g., dial-up, infrared, wireless, satellite).

Connections are used to model network topology, where they show which nodes can communicate. Connection characteristics (e.g., bandwidth, latency, availability, security) are important in assessing whether a network can satisfy service-level requirements of the various interactions carried.

Data group An old term, interpreted as a deployment unit that contains the state of one or more components.

Deployment unit A deployment unit describes one or more components grouped together for deployment purposes. It is intended to simplify the activity of placement since a deployment unit represents grouped components being considered for placement. A deployment unit has no effect on the structure, behavior, or performance of the software. During placement, deployment units are placed on nodes.

In practice it is often necessary to regard the execution aspect of a component (the place where the component executes) as being in a different deployment unit from the data storage aspect. In some environments (e.g., Java** applets), it is also important to distinguish between the deployment unit where the code of a component is installed and the deployment unit (on a different node) where the component executes.

Design pattern A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.[6]

Domain A domain is a subject area that defines a context for analysis and description of some aspect of an IT system. Domains are described by architectural templates, consisting of a set of collaborations (and their associated component roles or interfaces). See also architectural template.

Elaboration point An elaboration point is a milestone in the development of an artifact at which significant decisions are documented. Elaboration points are often used:

- As formal review points for an artifact
- As points at which different artifacts are synchronized

Many elaboration points are possible depending on the artifact and the problem at hand, for example:

1. An initial elaboration point is the initial and most abstract elaboration point, not reflecting technology and other constraints.
2. Intermediate elaboration points are where selection criteria for products have been defined, and fully specified operations have been defined for interfaces.
3. A physical elaboration point is where actual products and final hardware topology have been selected and placed, and to which compilers, languages, and libraries have been committed.

The elaboration points for different artifacts do not necessarily correspond.

Framework A set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.[6]

Functional aspect The functional aspect of an IT system is concerned with the functionality of collaborating software components. The functional aspect is expressed as one or more component models, which represent the static structure and dynamic behavior of the components in the system.

Components are defined in terms of interfaces and operation signatures. Structure is defined in terms of component composition and component usage relationships. Behavior is defined in terms of component collaborations, expressed as sequence diagrams.

The component models reflect the need to satisfy service-level characteristics.

Interaction An interaction specifies the details of the communications that should take place between two components in accomplishing a particular scenario. An interaction is defined in the context of a collaboration and describes which requests should be sent and their sequence. An interaction can be thought of in terms of a contract between two components and is often specified in terms of a protocol.

Interactions may have service-level requirements (e.g., data flow rates, availability) various types of the interaction mode (e.g., synchronous/asynchronous/batch, client-server/peer-to-peer), and order of requests defined in protocols like the Transmission Control Protocol/Internet Protocol (TCP/IP).

Examples of interactions include: a client/server interaction between a GUI on a client workstation and a relational database on a server through a TCP/IP socket, the communication between a shallow proxy and the underlying (remote) object.

Interface An interface specifies a set of operation signatures that are made externally available by a component to other components. The state and functionality of a component is hidden, and is only made externally accessible through the interfaces of the components. The interfaces are the only "public" or "visible" part of the component.

An interface may be provided by several components and used by several components. Interfaces are sometimes referred to by the related term API (application programming interface).

IT system An IT system is a combination of hardware, software, and documentation that implements and describes a business solution.

Location A location is a type of geographical area or position. Strictly speaking, it would be more accurate to call it "location type," as each location in the operational model (e.g., regional office) is not a specific grid reference, but a type of location, of which there may be several instances. Locations can be broad areas and contain more specific (sub) locations. For example, a location may represent a zone (e.g., central), a building (e.g., store, regional office), or a room within a building (e.g., server room).

Locations are used to represent the positioning of nodes and guide component and data placement decisions, as well as overall deployment considerations.

Network See connection.

Node A node represents a hardware platform (at some level of abstraction) onto which deployment units can be placed. Nodes are used to define required processing capabilities at locations, and (eventually) become detailed specifications for processors, memory, etc. A node may have characteristics such as memory, clock speed, and secondary storage capacity.

Nodes are commonly visualized in terms of diagrams showing network topology. Each node is at a location.

Operation signature An operation signature is a specification of a service offered by a component, i.e., a specific kind of request that can be made to a component.

An operation signature typically includes a description of the information that is passed along with the operation request and the information returned, together with any possible error situations that occurred while executing the request. For example, a TCP/IP connection operation can be represented by the following operation specification:

integer tcpip_accept_connection(s, &ns, wait_flag, timeout_value)

An interface consists of a set of operation signatures that are likely to be used in the same context.

Operational aspect The operational aspect of an IT system is concerned with the distribution of components across the geography of the organization in order to achieve the required service-level characteristics (performance, availability, etc.). It is also concerned with the necessary systems management functions and activities needed to maintain components (software distribution, responding to alerts, etc.).

The operational aspect is represented by one or more operational models, which show the type and location of hardware nodes, con-

nections, network topology, and placement of components or deployment units.

In order to emphasize spatial organization, the operational model does not show details of functional issues such as how software components collaborate.

Organizational unit An organizational unit is a group of people and resources with a specific business goal, which is related to other such groups via the organizational structure of a company.

Placement Placement refers to the activity of placing deployment units (and components) onto the network topology of nodes and connections in order to make functionality available at required locations and to satisfy service-level requirements. Where nodes are not predefined, the placement exercise leads to the identification and design of nodes and connections.

For example, a "clerk GUI" deployment unit may be placed on the clerk workstation, and an account data deployment unit may be placed with the DB2* (DATABASE 2*) deployment unit on the regional office data server RS/6000*.

The operational model is used to support and document the placement process.

Process group An old term interpreted as the execution role of one or more components mapped into a deployment unit.

Protocol A protocol extends the concept of an interface to include the allowable sequences of requests, possibly across many interfaces. It is defined in terms of interactions.

Reference architecture A reference architecture is one that has already been created for a particular domain of interest. It typically includes many different architecture styles, applied in different areas of its structure. See also architectural template.

Scenario A scenario is an instance of a use case. That is, a scenario is an execution of a use case under well-specified assumptions. A scenario is realized in the IT system by a collaboration. An example of a scenario is: "Use ATM 1234 to draw $100 from checking account 987654, where the account has no overdraft facility and has a previous balance of $105, and the transaction is successful."

Scenarios are used for:

- Validating and enriching the use case model
- Designing collaborations
- Prototyping (Scenarios can drive prototyping and are a useful way of defining prototype scope.)
- Test cases (Scenarios make good system and integration test cases.)
- User acceptance testing

Structuring In this context, structuring refers to the architectural activity of organizing the IT system into a set of interacting software components. Structuring addresses several considerations, including:

- Allocation of responsibilities to components in such a way that each component has a cohesive set of responsibilities and redundancy is avoided
- The partitioning of components to take into account distribution requirements
- Optimizing the component structure to satisfy service-level requirements such as performance
- The incorporation of reusable components or legacy systems into the design

Structuring plays a role in most of the design activities concerned with the functional aspect of an IT system. The component model supports and documents the structuring process.

Subsystem A subsystem is a grouping of components in an IT system.

A subsystem can be thought of as defined by an irregular line (e.g., a string) drawn around a subset of the components in a system. There are no restrictions on the subset.

Subsystems may overlap, i.e., a single component may be in two or more subsystems. A subsystem may contain other subsystems. A subsystem may span across nodes. A subsystem allows us to group a number of components for various reasons:

- Allocation of work to a development group: a development group is assigned one subsystem comprised of one or more components.
- Labeling major parts of an IT system on the grounds of the functionality offered by all of them: e.g., the imaging subsystem, the document-handling subsystem

This concept allows us to consider groupings of components without implying that these components are part of a larger component that offers services based on the services offered by the individual components contained in the set. A subsystem does not have any notion of encapsulation.

Technical reference architecture A technical reference architecture is a type of reference architecture that does not directly include structures of application (business) behavior. In other words, it can be used as a base architecture for several different application types. It nevertheless still applies only to a specific technical domain. For example, technical reference architectures or fragments of technical reference architectures exist today in the domains of distributed object systems (CORBA), compiler development, and the Internet (Web browser or server).

Use case A use case is an identifiable and externally observable behavior of the IT system. It is a pattern of usage that is initiated by an actor and that performs or aims to perform some useful work. A use case represents a dialog between an actor and the system. For example, "Draw funds from checking account" is a use case.

A template with standard sections (e.g., actors, preconditions, steps) is used to structure the description of a use case. During the definition of components, use cases are also used to describe behaviors that are internal to the IT system but are externally observable behaviors of the component.

Walkthrough A walkthrough is a description of the flow of a scenario starting from a user all the way through the system and back to the user. It corresponds to a collaboration between placed components. These textual descriptions may be augmented by sequence diagrams, which show the flow of messages between deployment units. An example of a walkthrough is the handling of a phone call in a call center application.

Walkthroughs are used to validate the operational model and to ensure that service-level requirements are satisfied.

Cited references and notes

1. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley Publishing Co., Reading, MA (1998).
2. In this context, infrastructure consists of those components (usually, but not exclusively, application-independent services) that may be used by many applications.
3. The Object Management Group, http://www.omg.org/. The UML notation and semantics guides are available from this site.
4. M. Fowler, K. Scott, and G. Booch, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley Publishing Co., Reading, MA (1997).
5. Although the majority of components will be software, it is sometimes necessary to model hardware as components. This will generally be the case if software components use specialized hardware interfaces directly (for example, an encryption device, a pager, or an interactive voice response unit).
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).
7. The Workflow Management Coalition is a nonprofit, international organization of workflow vendors, users, analysts, and university or research groups. Its mission is to promote and develop the use of workflow through the establishment of standards for software terminology, interoperability, and connectivity among workflow products. See http://www.aiim.org/wfmc/ for more information.
8. It is expected that these elaboration points will be identified in a later version.

Accepted for publication October 4, 1998.

Robert Youngs *IBM United Kingdom Ltd., 1 New Square, Bedfont Lakes, Feltham, TW14 8HB England (electronic mail: robert@uk.ibm.com).* Mr. Youngs is a Consulting IT Architect in the IBM United Kingdom Object Technology Practice. He has been a member of the ESS Technical Architecture team since 1996, and now works on the ESS Operations team. He was a member of the SI/AD Architecture Description Standard project and is the architect of the ESS tool. He has worked as an enterprise systems engineer and a technical consultant for large IBM enterprise clients in the banking, government, and utilities industries, as a specialist in enterprise systems software and hardware, and as a project manager. Mr. Youngs was a member of the project that created IBM's first infrastructure design method (End-to-End Design). He holds an M.A. in mathematics from Cambridge University.

David Redmond-Pyle *PostModern Solutions Ltd., Sunny Croft, Tarvin Road, Manley, Cheshire, WA6 9EW England (electronic mail: david@postmod.com).* Mr. Redmond-Pyle is Technical Director of PostModern Solutions, and is currently working with the IBM United Kingdom Object Technology Practice as a consultant on method development. He was a core member of the SI/AD Architecture Description Standard project and contributed to the development of architecture work product descriptions for the new SI/AD method and to methods related to ESS. He previously worked as Chief Methodologist at CASE and as methodology specialist at LBMS, Inc., where he was responsible for developing LBMS's client/server development method. At LBMS he worked on a wide variety of methods and tools, including DSDM/Internet, software reuse methods, formal specification in Z, designing parts of the Systems Engineer CASE tool, and working with the OMG and the Workflow Management Coalition. He has also collaborated with Hewlett-Packard Company on their Fusion2 OO method. Mr. Redmond-Pyle has published numerous articles and an influential book on the design of graphical user interfaces (the GUIDE method).

Philippe Spaas *IBM United Kingdom Ltd., 1 New Square, Bedfont Lakes, Feltham, TW14 8HB England (electronic mail: philippe_spaas@uk.ibm.com).* Mr. Spaas is an IT Architect in the IBM United Kingdom Object Technology Practice. He has worked as an application architect on various projects in the financial sector. He was a member of the ESS Application Architecture team in 1997 and participated in the Architecture Description Standard project in 1998. He also made contributions to the SI/AD methods initiative. Mr. Spaas holds a degree in applied economics from the Katholieke Universiteit Leuven (Belgium) and an M.B.A. from Cornell University.

Ed Kahan *IBM Global Services SI/AD National Practice, 315 East Robinson Street, Orlando, Florida 32801 (electronic mail: Ekahan@us.ibm.com).* Mr. Kahan is a certified Executive Consultant and a member of the national Systems Integration/Application Development (SI/AD) practice. Mr. Kahan is the team leader for the Architecture Description Standard development and the infrastructure architecture and design methods used by IBM. He is also working on ESS and architectures for asset harvesting and reuse. Prior to his current assignment in the national practice, he consulted with IBM clients in the telecommunications, transportation, banking, utilities, and petrochemical industries in application and IT architectures development. Mr. Kahan was a founding member of the consulting group in Florida. Prior to joining IBM, Mr. Kahan worked in signal analysis, acoustics, and vibration instrumentation research at Bruel & Kjaer Instruments, Denmark.

Reprint Order No. G321-5696.