Documenting software architecture, Part 4: Develop the functional model

Moving from the abstract to more detailed constructs

Tilak Mitra
Certified Senior IT Architect
IBM

29 July 2008

In this series, learn why and how you should document software architecture. In this article, learn how to develop and document the macro-level design artifacts of the functional aspects of your system's architecture. The functional model view addresses the techniques you can use to decompose the problem domain into a set of architecture artifacts. Learn to build upon them incrementally to form more detailed constructs. The three common levels of elaboration—logical level, specification level, and physical level—are also discussed.

View more content in this series

Introduction

In Part 1 of this series, you learned about the importance of a disciplined approach to documenting software architecture and about mechanisms that can capture the architectural artifacts used in a typical development process. Part 2 focused on the system context, which is the first important architecture artifact, and how to document system context information with diagrams and information flows. In Part 3 you learned how to develop and document the architecture overview for your system or application.

This article focuses on the functional architecture components of the system. Learn how the architecture building blocks are decomposed into design-level constructs that collectively realize the functional requirements of the application or system to be built.

Each architecture building block describes, at a high level, the capabilities of a component in the context of the entire solution. The components help define the architecture blueprint and are broadly characterized as functional or operational in nature. But they are too coarse-grained to be directly consumable by the development teams to transform them into code.

This article also provides recommendations on how to transform the functional components into macro-level design artifacts. The focus is on documenting the macro-design artifacts, also known

as the functional model. Learn about the various levels of a functional design model, going from general to more specific, and how to document the various levels.

Functional model overview

Functional modeling, also called component modeling, focuses on building the functional architecture of the system by breaking the problem domain into a set of non-overlapping and collaborating components. The IBM Rational® Unified Process® (RUP®) states that modeling should be done at least at two different levels: the analysis level and the design level. The main difference between the two levels is the degree of specificity that they illustrate. The design models build upon the analysis models by adding details that contain enough information to facilitate the third level of modeling—the implementation model. Analysis and design models can be called a macro design, while implementation modeling is a micro design.

This article discusses the elements of macro design, which can be considered part of a functional view of the architecture. It builds upon the principle that analysis modeling and design modeling fall under the purview of architecture. Finer-level detailing of design models and implementation models falls under the purview of design.

Documenting the functional model

The functional model is built in three iterations, with designs at the:

- Logical level
- Specification level
- Physical level

With each successive iteration, you move from higher levels of abstraction to more specific design and implementation artifacts. The rest of this article discusses the three iterations.

Logical-level design

Some schools of thought prefer to keep the logical level of design very conceptual, with information only in terms of business concepts that have no relevance to IT. That level of conceptualization is acceptable, but now there's a discipline called '"business architecture," which focuses on defining the business domain through a set of business-oriented concepts and constructs. Business architecture is becoming a common technique to define the business aspects of an enterprise architecture. This article won't address the conceptual constructs in the logical-level design of the functional architecture.

Before learning about the logical constructs of the functional model, you need to first understand the traceability of artifacts to the business domain. Being able to trace IT architecture constructs to the business domain is very important. You can then ensure that architecture documents are coherent and are aligned to the business drivers, goals, and problems that the architecture will solve. Business analysts analyze the business domain and capture the business requirements in a technology-neutral format. They try to capture "what" should be built, while leaving "how" it should be implemented to the IT architects, designers, and implementation team.

Typically, business analysts identify a set of business domains from within the enterprise business model that span or are touched by the problem to be solved. Each business domain, such as accounting, order fulfillment, and so on, is further decomposed into a set of functional areas. The domain of functional area analysis involves the business analyst identifying a set of business functions from within a business domain, which is logically cohesive enough to be a single, functional unit. The business analyst categorizes by functional area the requirements that are in the scope of the initiative and which are then handed over to the IT team.

Subsystem design

A *subsystem* is a first-class IT construct that is a direct rendition of the functional areas. A functional area in the business domain can be represented and realized by one or more IT subsystems. A subsystem is a grouping of cohesive components that tend to change together, so that the changes (in the form of enhancements or fixes) can be controlled so that their effects are localized within a subsystem boundary. Just as functional areas are mapped to and decomposed into IT subsystems, the business functions are realized by one or more IT functions. These IT functions are logically grouped, encapsulated, and implemented as a single unit. That unit is the IT subsystem. You can implement IT functions using a group of software components, so a subsystem is a grouping of software components.

The IT functions are exposed by a set of interfaces at the subsystem level. One or more software components inside the subsystem implements each interface. A subsystem groups together components that tend to change, so the changes (in the form of enhancements or fixes) are controlled and their effects are localized within a subsystem boundary. Subsystems foster parallel development, with the subsystems and their interfaces designed beforehand. Implementation teams develop the internals of the subsystem while adhering to the external interface contracts.

The first activity in this phase is to identify IT subsystems, so they can then be documented. For each subsystem, each of its high-level interfaces also needs to be identified and documented. I recommend that you document the following artifacts for each subsystem:

Subsystem ID
: Provides a unique ID for each subsystem so that it's easy to reference each of them in the design.

Subsystem name
: Provides a name for each subsystem, such as accounts management, transaction management, and so on.

Functions
: A list of IT functions that the subsystem exposes through its internal implementation. I recommend identifying this set by analyzing the system use cases, grouping them logically, and assigning them to an identified subsystem.

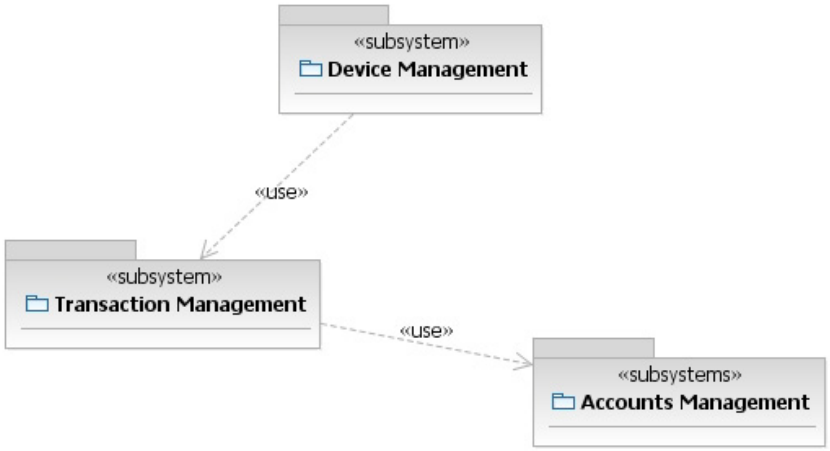
Interfaces
: A list of interfaces that the subsystem supports or exposes. For example, in an accounts management subsystem, an interface might be called "withdrawal." At this level, a textual description of the interface and its operations would suffice.

Your template will look something like this:

Artifact identifier	Example
Subsystem ID	SUBSYS-01
Subsystem name	My Subsystem
Function(s)	F1,F2
Interface(s)	I11, I12, I21

You should produce a UML representation of the subsystems and their interdependencies as part of documenting the design artifacts at the logical level. Figure 1 shows an example.

Figure 1. Subsystems and their dependencies



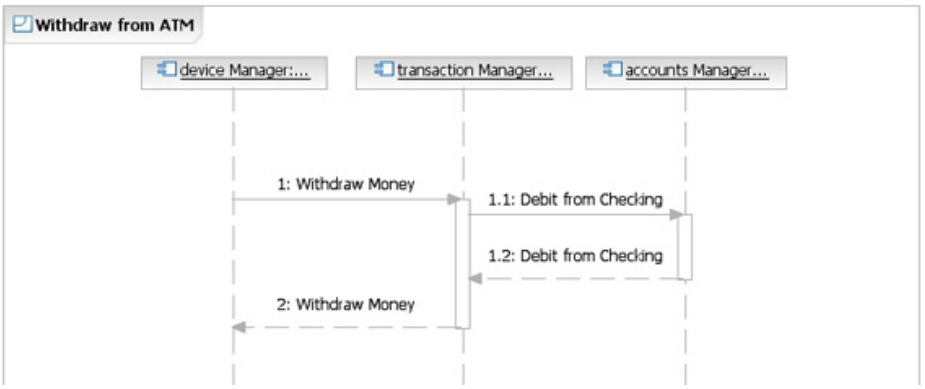
Component design (logical level)

After you have identified the subsystems and documented their responsibilities, the next step is to identify a set of high-level software components that collectively realize the interfaces exposed by the subsystem. An IT subsystem is a first-class manifestation of the functional area into the domain of IT. As such, the IT functions within a subsystem can be partitioned based on the core business entity that they address. For example, for the accounts management subsystem there can be a software component that caters to savings accounts, while another one focuses on implementing the features of the checking account.

After the components are identified at a logical level, you need to identify the architecturally significant use cases. Analyze the use cases, then choose subsets that are significant from an architecture standpoint. For each of the architecturally significant use cases, you can use component-interaction diagrams to elaborate on how the use case may be realized through the functions that are exposed by the components you have identified.

A collaboration diagram shows how components interact by creating links between the components, and by attaching messages to these links. The name of the message denotes the intent of invoking the component when it interacts with the involved component. At this logical level, you can think of the messages as pseudo operations on the components. These pseudo operations manifest themselves as the responsibilities of the component. Figure 2 shows a sample collaboration diagram.

Figure 2. High-level component interaction



I recommend that you document the following information for each component and subsystem:

Subsystem ID
: Denotes the unique identifier of the subsystem to which the component that is addressed belongs.

Component ID
: Provides a unique identifier (ID) for the component.

Component name
: Provides the name for the component.The name should be intuitive, based on the business entities that the component focuses upon.

Component responsibilities
: A textual description of a set of responsibilities that are assigned to the component and expected to be realized.

Your template should look something like this:

Artifact identifier	Example
Subsystem ID	SUBSYS-01
Component ID	COMP-01
Component name	My Component
Component responsibilities	F1, F2, F3

Specification-level design

Specification-level design is another name for detailed design. You build upon the logical level constructs by adding details for each of the components until:

- The interfaces for each component are well-defined.
- The data elements owned by each subsystem are identified and detailed.
- The component's responsibilities are fleshed out in more detail.

I recommend that you take four main steps during the specification-design exercise. The rest of this section discusses each of the steps.

1. Component responsibility matrix

In this step, you build upon the initial matrix that was developed during the logical-level design. The main additions to the existing matrix are a more detailed and refined set of component responsibilities.

The existing responsibilities were identified based on the functional specifications (obtained by analyzing the use cases). The nonfunctional requirements (NFRs) were left out of the application. The NFRs are usually documented in a separate document artifact. You should do a careful analysis of each NFR, identify a component, and assign the fulfillment of the responsibility to the identified component.

Business rules have now become a mainstream component of any application. Business rules are an expression of the business decisions in the domain of IT programming. Business rules and policies change so frequently that enterprises must sense and respond to changes in the marketplace and quickly adapt their IT systems to maintain a competitive and differentiated advantage.

Business rules cannot be embedded into the core programming logic. Embedding business rules will result in legacy systems that are difficult to change. Business rules need to be externalized so that they can be changed at run time. However, such rules need to be identified and assigned as a responsibility of a component. Such components may leverage the capabilities of a business rules engine to realize the parts of its responsibility that are identified from the business rules catalog. In some cases, an entire component may be dedicated to interface with the APIs exposed by the rules engine.

At this point, the component responsibility matrix is augmented with the previous documented details. A snippet of the template might look like this:

Artifact identifier	Example
Subsystem ID	SUBSYS-01
Component ID	COMP-01
Component name	My Component
Component responsibilities	F1, F2, F3 NFR 001 - Description and linkage to the NFR document NFR 005 BRC 001 - Description and linkage to the business rules catalog (BRC) document BRC 005

2. Interface specification for components

During the logical-level design, you have identified interfaces and associated them with subsystems. Now that you have a detailed exposition of the responsibilities that each component is expected to perform, you can focus on interface definition and design.

An *interface* is the mechanism by which a component exposes its functions to the outside world. In technical terms, an interface is a collection of operations or methods. Interface

specification involves the challenge of identifying operations and grouping them into interface boundaries.

To start, you should analyze each use case that is owned by a component, keeping the key factors of complexity and cohesiveness in mind. System-use cases that are very atomic in nature, such as retrieving a customer profile, should be categorized as an operation (on an interface).

Often, though, use cases are documented at various and sometimes inconsistent levels of granularity. Some use cases are written so that the main flow is to create a particular entity, where its alternative flows are to update or delete the entity. Such use cases need to be tackled in one of two ways:

- Refactor the use case, and break it down so the main operations on the entity are in their separate use cases.
- Treat the use case at the level of an interface, where the interface is a collection of a set of logically cohesive operations.

After you have identified the operations through use case analysis, interfaces should be formally identified. The approach I recommend, as mentioned earlier, is to group operations that enjoy logical cohesiveness, work on the same set of business entities, and are mutually exclusive from the rest of the operation set. A logical grouping of such operations is defined as an interface, which is a first class construct in Object Oriented Programming. This exercise identifies a set of such interfaces that are then exposed by a given component.

The first step of the interface design is to document and model the interfaces and their operations with the proper signature and parameter list. For example:

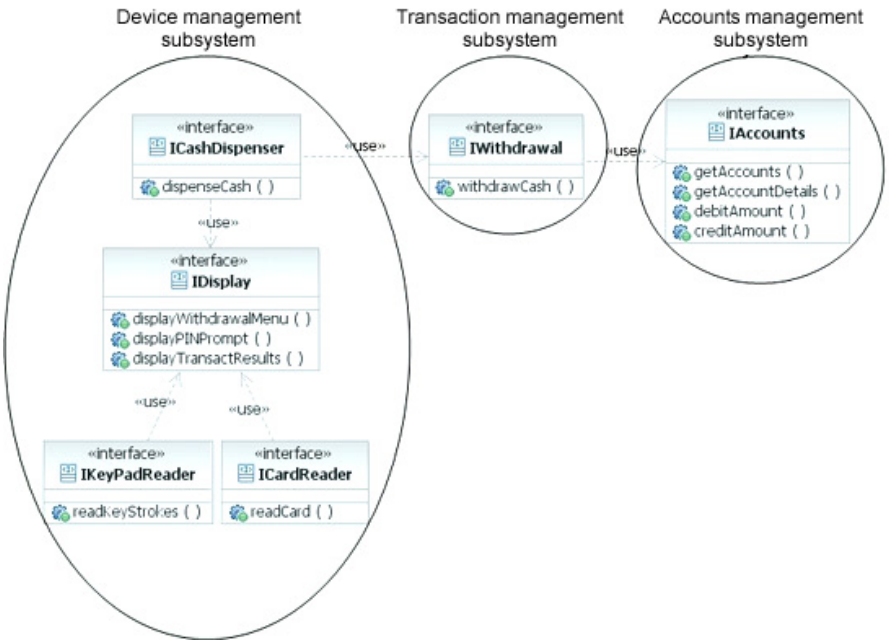
Artifact identifier	Example
Component ID (it belongs to)	COMP-01
Interface name and ID	My Interface, INT-001
Interface operation(s)	Provide signatures for each of the operations along with their descriptions

The second part of this task is to identify how interfaces are dependent on other interfaces. There are two types of interface dependency: when they depict the dependencies on interfaces that are within a subsystem boundary, and when they depict dependencies on interfaces that are exposed by different subsystems. The dependency is usually documented as a UML class diagram, where each class is stereotyped as an interface, and association lines are used to explicitly depict the dependencies.

Complete documentation takes this one step further and provides a textual description that qualifies and elaborates on the exact nature of the dependency. For example, Interface1::Operation11 has a post-condition dependency on Interface2::Operation21. This is the recommended approach, but the realities of time and cost can sometimes limit our freedom to do so.

Figure 3 shows a typical interface-dependency diagram. The subsystem boundaries for the interface are also shown.

Figure 3. Interface dependency inside and between subsystems



3. Identifying and associating data to subsystems

Until now, the focus has been on component responsibilities. One of the important tenets of component design is to identify the data entities that are owned by a subsystem and how they are used by the components within the subsystem to realize their responsibilities.

The logical data model is used as an input to this task. The logical data model identifies the core business entities that are in the scope of the system to be built. A subsystem has a set of responsibilities to fulfill that are implemented by the components encapsulated within the subsystem. The components expose the responsibilities through the interfaces, and more specifically, through the interface operations.

Each of the interface operations requires data to be operated upon to realize the functions. The parameters on the interface operations are suggestive of a grouping of data entities that are used in close conjunction. Use this simple guideline to help identify the data entities and associate them to subsystems:

1. Analyze the parameter list on the interfaces.
2. Map the parameters to the closest business entities or data types in the logical data model.
3. Repeat steps 1 and 2 for each of the interfaces on a component.
4. Keep a running list of the data types that are identified.
5. Repeat steps 1 - 4 for each of the components within a subsystem.
6. Consolidate the list of data types identified in steps 1 - 5.

Draw a boundary around the identified data types from the logical data model, and associate the data types to the subsystem. After you do this for all the subsystems, you might have the common situation that a few data types are identified as belonging to more than one subsystem. For such data types, architecture rationalization is required to:

- Analyze and assess which subsystem performs the primary operations on the data type.
- Determine the subsystem that will be primarily responsible to own the data type.

With data ownership defined, the interface parameters are now refined to use the actual data types to define the inputs and outputs. You should draw a view of interface dependency, using standard UML notations, which explicitly identifies the dependencies of interfaces to the data types. A picture really is worth a thousand words when documenting software architecture artifacts. In this case, it's better to use UML model artifacts to depict the subsystem and component ownership of data entities. Typically, providing textual descriptions for each of the data types isn't required. You would refer to the logical data model documentation to get such detailed descriptions.

4. Component-interaction diagram

During the logical-level design, you developed and documented a high-level component-interaction diagram. At that level, the components were invoked through pseudo operations only. From then until now, a lot of detailed specifications have been developed, the component responsibility matrix was elaborated, the interfaces were specified, and the data types were identified and assigned ownership to subsystems.

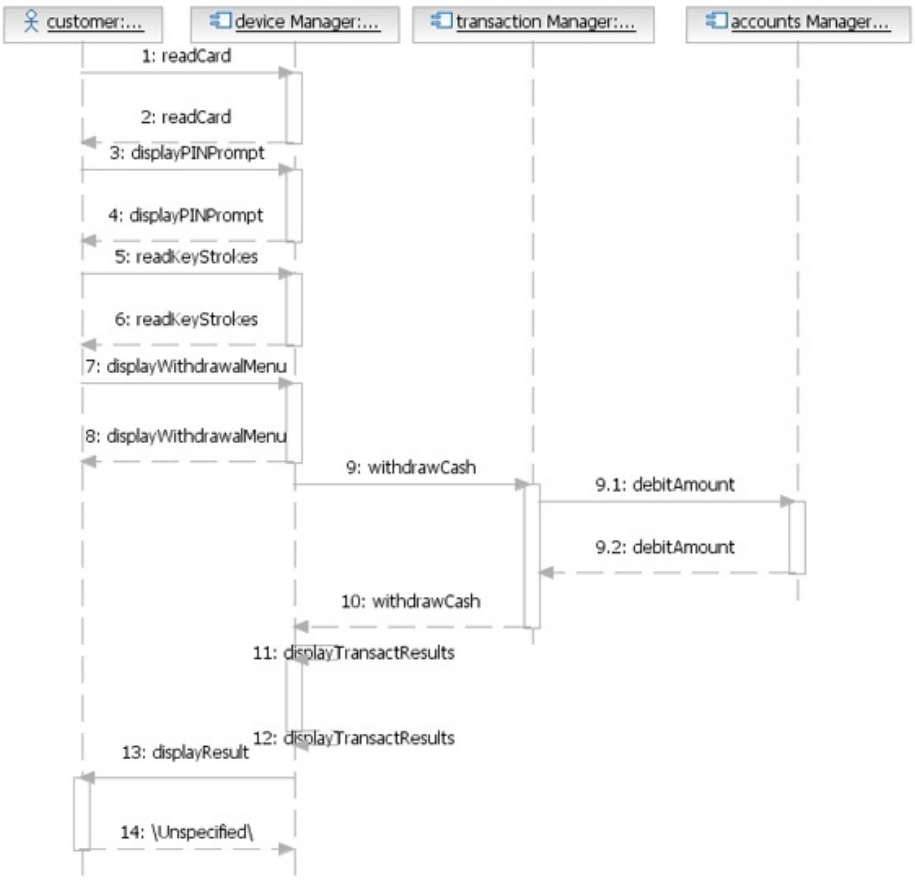
Use cases are chosen that exploit each operation on the interfaces that the components expose. The architecturally significant use cases might need to be supplemented with other use cases to provide coverage for all the operations on each of the interfaces. For each use case, use UML sequence diagrams to draw the component-interaction diagrams. Each component-interaction diagram starts from an originating requester (actor), invokes specific operations on a series of components that collectively realize the use case, and returns the result to the originating requester. Figure 4 shows an example.

For each of the UML sequence diagrams, a textual description of the step-by-step invocation of the operations on the components is also documented.

At this stage, each subsystem is well-defined, with each component having a set of responsibilities that are in turn exposed through one or more interfaces. Each interface is specified with a set of operations, and each operation is defined through a list of input and output parameters. Each parameter is mapped to a data type that is owned by the subsystem.

Subsystems also require refinement and refactoring. If a subsystem has grown to take up too much responsibility, it might be too complex to implement. If it looks to be too thin in features, you may need to consolidate and merge it with another related subsystem. And, not all subsystems need to be custom-built. Some represent existing assets or products (such as a business rules engine), and using such subsystems is an opportunity to foster reuse.

Figure 4. Component-collaboration diagram for the use case "Withdraw from ATM"



This completes the recommended steps for documenting the specification-level design artifacts.

Physical-level design

The physical-level design essentially revolves around distributing the application and subsystems on preliminary nodes, so they can be installed and run on physical hardware nodes that collectively represent the infrastructure topology of the system. The physical-level design is also used to influence technical infrastructure and other middleware components and subsystems.

The subsystems need to be deployed on a set of hardware nodes. The system's hardware configuration has different characteristics and specifications for each of the nodes.

- Some might be tuned for customer interaction through high-end user interfaces
- Others may be tuned for high-volume transactional systems.
- Some require hardware-level security.

The NFRs that involve availability, performance, reliability, and other qualities have a strong influence on the selection of hardware and middleware that must support the applications that reside on top of them. New subsystems may also need to be identified based on the choice of

technology. For example, to support distributed objects, you can use DCOM, if it is based on .NET technology, or remote method invocation (RMI) over IIOP if it is based on Java™ 2 Enterprise Edition (J2EE). Such technology-specific tenets can also be identified as their own subsystems. When new subsystems are identified in this manner, a technology agnostic interface is defined, through which the other subsystems in the application domain get to interact. This provides a facade over the technology-specific implementations and introduces enhanced reusability in the system's design. You should strive to achieve reusability at each layer of the architecture (see Part 3 of this series).

Once you have analyzed and understood the characteristics of the node, and have identified new subsystems (if required), the main activity is to match subsystems (based on functional and technical characteristics and NFR requirements) with the hardware node that's the best candidate to realize the capabilities of the subsystem. After this analysis, each subsystem can be placed on a node in the operational topology of the system. A node can typically be a container of multiple subsystems, though there may be specialized nodes that are fine-tuned for only a specific type of subsystem.

> A detailed treatment of nodes, their description, configuration, dependency on other nodes, and protocols used for internode connectivity are in another category of the operational view. This will be covered in an upcoming article.

There is some overlap of the physical design artifacts with the operational model view of software architecture.

There are various schools of thought that try to define and document software architecture in different ways. One school considers physical- level design as more of the micro design of the components. Micro design is the domain in which a component is considered to be the highest level of abstraction. Each component is broken down into a set of participating classes that collectively realize the operations that the component exposes through its interfaces. The designer applies the well-known and proven design patterns to solve a specific pattern of a problem. Patterns can be combined to develop composite design patterns that together solve a given problem domain within the component. Detailed sequence diagrams are used to elaborate the dynamic nature of how each operation (on the interface) is implemented through the participating classes in the class model. Such detailed design is performed for each of the components in the scope of the application.

I consider these detailed design activities to be under the broader discipline of design, and not necessarily architectural in nature. This series is purposely not trying to illustrate documentation of design-level artifacts.

Conclusion

The functional model is one of the most important domains of software architecture. Some architects focus only on this domain and consider it to be their software architecture. The functional model view addresses the architectural techniques used to:

- Decompose the problem domain into a set of architecture artifacts.

- Progressively build upon the artifacts by incrementally moving from the abstract to more detailed architectural constructs.

This article discussed the functional model view of software architecture. You learned about the three most commonly used levels of elaboration: the logical-level, specification-level, and physical-level design. You learned what to document and how to document the artifacts in each level of abstraction.

I recommend that you read the first few articles in the series to learn about the essence of software architecture, the why and how of documenting, and the dimensions of architecture that the other views address. Stay tuned for upcoming articles that will focus on the dimensions that haven't been covered yet.

Resources

Learn

- Check out the other parts of this series:
  - Part 1, "What software architecture is, and why it's important to document it"
  - Part 2, "Develop the system context"
  - Part 3, "Develop the architecture overview"
- Get the RSS feed for this series.
- Learn more about SOA in *Executing SOA: A Practical Guide for the Service-Oriented Architect*, a recently published developerWorks series book that Tilak coauthored. Use coupon code IBM3748 for a 35% discount.
- Read a compendium of published software architecture definitions.
- *The Unified Software Development Process* by Jacobson, et al provides an excellent treatment of how Rational Unified Process (RUP) is used to document software artifacts.
- *A Component-Based Development Pattern Language*, Arsanjani A., European Conference on Pattern Languages of Programming, 2001.
- In the Architecture area on developerWorks, get the resources you need to advance your skills in the architecture arena.
- Browse the technology bookstore for books on these and other technical topics.

Get products and technologies

- Download IBM product evaluation versions and get your hands on application development tools and middleware products from IBM® DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Tilak Mitra

Tilak Mitra is a Senior Certified Executive IT Architect in IBM. He specializes in SOAs, helping IBM in its business strategy and direction in SOA. He also works as an SOA subject matter expert, helping clients in their SOA-based business transformation, with a focus on complex and large-scale enterprise architectures. His current focus is on building reusable assets around Composite Business Services (CBS) that has the ability to run on multiple platforms like the SOA stacks for IBM, SAP and so on. Tilak recently coauthored a developerWorks series book: *Executing SOA: A Practical Guide for the Service-Oriented Architect*, available in Resources. He lives in sunny South Florida and, while not at work, is engrossed in the games of cricket and table tennis. Tilak did his Bachelors in Physics from Presidency College, Calcutta, India, and has an Integrated Bachelors and Masters in EE from Indian Institute of Science, Bangalore, India. Find out more about SOA at Tilak's blog. View Tilak Mitra's profile on LinkedIn or e-mail him with your suggestions at tmitra@us.ibm.com.