Elements of Service-Oriented Analysis and Design

An interdisciplinary modeling approach for SOA projects

Olaf Zimmermann (ozimmer@de.ibm.com), Senior IT Architect, IBM, Software Group
Pal Krogdahl (pal.krogdahl@se.ibm.com), Solution Architect, IBM, Software Group
Clive Gee (clive@us.ibm.com), Senior Solution Architect, IBM, Software Group

Summary:

Experience from first Service-Oriented Architecture (SOA) implementation projects suggest that existing development processes and notations such as Object-Oriented Analysis and Design (OOAD), Enterprise Architecture (EA) frameworks, and Business Process Modeling (BPM) only cover part of what is required to support the architectural patterns currently emerging under the SOA umbrella. Thus, there is a need for an enhanced, interdisciplinary service modeling approach.

In a recent interview at Info World (see Resources), Grady Booch stated that "the fundamentals of engineering like good abstractions, good separation of concerns never go out of style", but he also pointed out that "there are real opportunities to raise the level of abstraction again." Past experience indicates that the abstraction level has to be raised up to the business domains a company deals with, taking the entire enterprise IT landscape into account.

SOA, for example, as introduced by Mark Colan in the article, "Service-Oriented Architecture expands the vision of Web services, Part 1", is an emerging architectural style for crafting next-generation enterprise applications. While the SOA approach strongly reinforces well-established, general software architecture principles such as information hiding, modularization, and separation of concerns, it also adds additional themes such as service choreography, service repositories, and the service bus middleware pattern.

A structured approach or analysis and design method is required to craft SOAs of quality. As none of the existing approaches met the authors requirements on recent SOA projects, they suggest combining elements from well-established practices such as OOAD, EA, and BPM, complementing them with innovative elements upon demand.

Introduction

The basic concepts of Service-Oriented Architectures (SOAs) and Web services are becoming part of our everyday language and recognized as a suitable architectural style for crafting modern enterprise applications. In this context, the underlying issues of: what makes good services are becoming increasingly critical for ensuring the successful implementation of SOAs.

Existing modeling disciplines such as Object-Oriented Analysis and Design (OOAD), Enterprise Architecture (EA) frameworks, and Business Process Modeling (BPM) provide us with high-quality practices that can go a long way in assisting with the identification and definition of appropriate abstractions within an architecture. However, experience shows that these practices fall short when being applied independent of each other.

In this article, we will investigate suitable elements from OOAD, EA, and BPM. We will also motivate the need for a hybrid approach that combines elements of all of the disciplines, with a number of distinct, new elements. The resulting, interdisciplinary OOAD method facilitating successful SOA deployments, which we refer to as Service-Oriented Analysis and Design (SOAD), has yet to be formally defined. We merely take first steps into the SOAD space.

The concept of service-orientation

In anticipation of the discovery of new business opportunities or threats, the SOA architectural style aims to provide enterprise business solutions that can extend or change on demand. SOA solutions are composed of reusable services, with well-defined, published and standards-compliant interfaces. SOA provides a mechanism for integrating existing legacy applications regardless of their platform or language.

Conceptually, there are three major levels of abstraction within SOA:

- *Operations:* Transactions that represent single logical units of work (LUWs). Execution of an operation will typically cause one or more persistent data records to be read, written, or modified. SOA operations are directly comparable to object-oriented (OO) methods. They have a specific, structured interface, and return structured responses. Just as for methods, the execution of a specific operation might involve invocation of additional operations.
- *Services:* Represent logical groupings of operations. For example, if we view *CustomerProfiling* as a service, then, *Lookup customer by telephone number*, *List customers by name and postal code*, and *Save data for new customer* represent the associated operations.
- *Business Processes:* A long running set of actions or activities performed with specific business goals in mind. Business processes typically encompass multiple service invocations. Examples of business processes are: *Initiate New Employee*, *Sell Products or Services*, and *Fulfill Order*.

In SOA terms, a business process consists of a series of operations which are executed in an ordered sequence according to a set of business rules. The sequencing, selection, and execution of operations is termed service or process choreography. Typically, choreographed services are invoked in order to respond to business events.

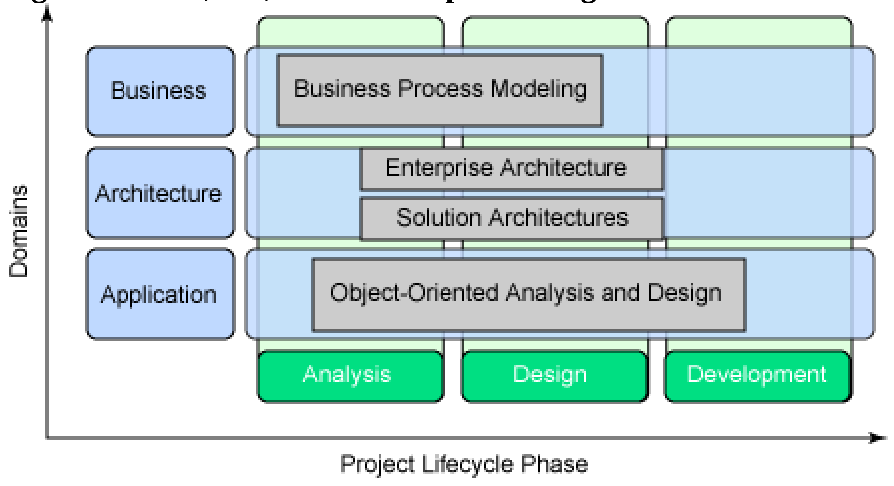
From a modeling standpoint, the resulting challenge is how well-designed operation, service, and process abstractions can be characterized and constructed systematically. The related issues currently are among the most frequently discussed ones in industry and academia. We are not aware of any recent SOA project or workshop in which such service modeling aspects have not been a major topic, giving fuel for numerous debates. So let us take a closer look.

Why BPM, EA, and OOAD are not enough

Experience from early SOA implementation projects suggests that existing development processes and notations such as OOAD, EA, and BPM only cover part of the requirements needed to support the SOA paradigm. While the SOA approach reinforces well-established, general software architecture principles such as information hiding, modularization, and separation of concerns, it also adds additional themes such as service choreography, service repositories, and the service bus middleware pattern, which require explicit attention during modeling.

Figure 1 illustrates where the existing EA, BPM, and OOAD modeling approaches have their main application areas. It also gives us a good starting point for the following discussion of SOAD. The horizontal axis in the figure is the project lifecycle phase; the vertical one differentiates between the different levels of abstractions or domains, on which modeling activities typically reside.

Figure 1. BPM, EA, and OOAD positioning



The SOA vision is accepted rather easily, as its technical foundation is well-known. For example, applying general software architecture principles and OO techniques is a valid start in any SOA effort. However, as already stated, the question most frequently asked by early adopters is how to identify the right services. As stated earlier, OOAD, EA, and BPM cannot supply a satisfying answer when applied in isolation from each other, as we will explain now.

The OOAD methodology introduced in the seminal books from Booch and Jacobson (issued about a decade ago), provides an excellent starting point in defining SOAs. As such, OOAD is concerned with micro-level abstractions such as classes and individual object instances, although applying OOAD techniques and the Unified Modeling Language (UML) notation on the architectural level has been common practice for many years. As a standalone use case model is frequently created per problem domain, and consequently, application development project, the enterprise-wide big picture gets blurred in many cases. Furthermore, for various reasons the use case models are not always synchronized with their BPM counterparts.

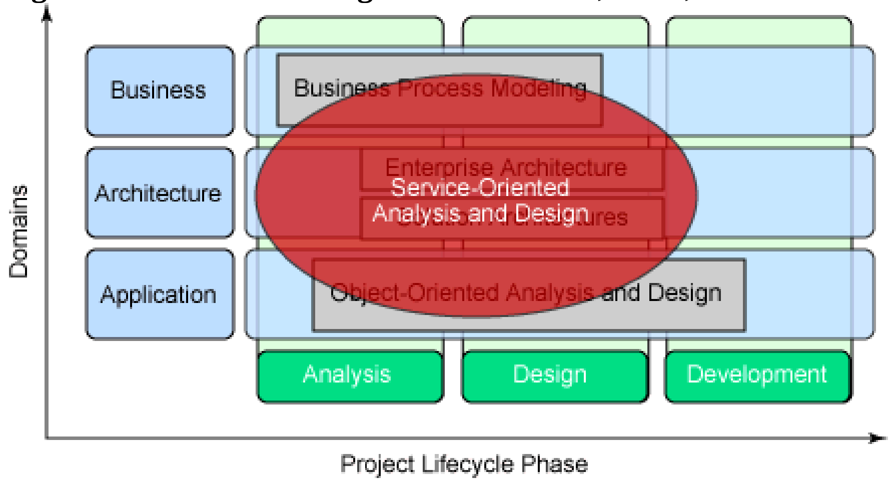
EA approaches, such as Feature-Oriented Domain Analysis (FODA), and Zachman add a city-planning level viewpoint on top of solution architectures, but do not address how enterprise-wide abstractions of quality facilitating reuse and longevity can be found.

While BPM approaches such as BPMI do provide an end-to-end view on functional units of work, they typically do not reach into the architecture and implementation domain. For example, until the arrival of languages such as the Business Process Execution Language for Web Services (BPEL), BPM notations were missing operational semantics. Moreover, we have seen many cases in which the process modeling and the development initiatives were separated from each other.

Finally, none of the existing disciplines address how existing applications can be enabled for SOA; a top-down process is employed most of the time. Existing systems typically hold large amounts of critical data and business logic, and cannot simply be replaced. Hence, a bottom-up analysis of these systems also has to be conducted in order to investigate wrapping and refactoring strategies. Taking existing applications into account, therefore, leads to a meet-in-the-middle process.

A hybrid SOAD modeling approach is required for these reasons. The approach comprises elements from OOAD, BPM, and EA in a best-of-breed fashion, and complements them with certain innovative elements. Figure 2 illustrates the SOAD assets (elements and techniques) for this new approach:

Figure 2. SOAD and its ingredients: OOAD, BPM, and EA



EA

Evolving enterprise applications and IT infrastructure into a SOA can be a major undertaking, affecting multiple lines of business and organizational units. Therefore, EA frameworks and reference architectures such as The Open Group Architecture Framework (TOGAF) and Zachman should be applied, striving for architectural consistency between individual solutions.

According to past experience, most existing EA frameworks have limitations in one or more areas. For example, no business-level process or service view can be obtained if the primary concern is how low-level building blocks, representing technical devices, interconnect at a macro-level. However, in a SOA context, this way of thinking must be changed to one centered around logical building blocks representing business services, concentrating on defining the interfaces and *Service Level Agreements* (SLAs) between the services.

Furthermore, many enterprise-level reference architectures and frameworks are rather generic, and do not reach down to the design domain. Such high-level architectures fail to give concrete, tactical advice for solution architects and developers; a fundamental gap between enterprise and solution architectures frequently is the consequence.

SOAD has to assist the SOA architect in defining a holistic, business-level view of the services landscape. This is something today's EA frameworks cannot provide without future SOA-specific enhancements; the *on demand Operating Environment">On Demand Operating Environment* (ODOE) is a major IBM initiative heading this direction.

BPM

BPM is a fragmented discipline in which there are many different styles, notations, and assets. For example, UML usage commonly extends from the application domain to the BPM level. Another frequently used technique is to define *event-driven process chains* representing conceptual process flows, as defined by Barker and Longman. This second technique uses a different notation other than UML.

Furthermore, there are many proprietary approaches such as BPM techniques that might be viewed as a competitive advantage by consulting firms and Enterprise Resource Planning (ERP) package vendors. The ARIS Implementation Platform® is an example of such an offering. Other approaches include: Line of Visibility Enterprise Modeling (LOVEM™) and the Component Business Modeling (CBM) initiative from IBM.

A recent trend is to define a standard way of representing executable flow models (for example, the Business Process Execution Language for Web Services (BPEL)). BPEL extends the reach of process models from analysis to implementation. Such executable models raise a whole set of new questions including:

- Which aspects should be described in BPEL, and which in WSDL? Where is the split between process model and more traditional programming models?
- How can aspects such as non-functional requirements and quality-of-service characteristics be incorporated into the models?
- What amount of logic is executed in programming language extensions of BPEL engines, as opposed to more traditional coding, for example, in J2EE?
- How can the quality of executable process models be assessed, and what are the best practices that apply?
- Which job role performs the BPEL flow engineering; is it a business expert (analyst) or a development role (software architect)?

All existing BPM approaches can be leveraged as a starting point for SOAD; however, they have to be amended with additional techniques for deriving candidate services and their operations from the process models. Furthermore, process modeling in SOAD must be synchronized with design-level use case modeling, and answers to the BPEL-related questions must be given.

OO versus sevice-oriented (SO) paradigms

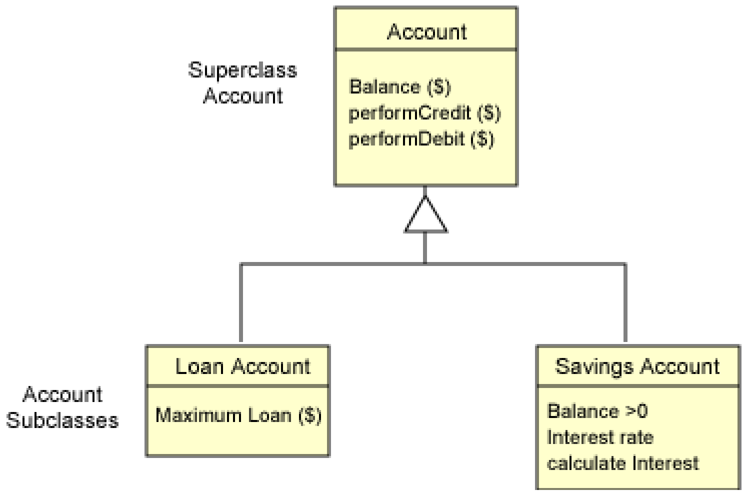
OO analysis is a very powerful and proven approach, and as such, SOAD should make use of OO analysis techniques as much as possible. To successfully apply OO analysis to SOA projects, you must examine more than one system at a time. Use case models will continue to play an important role. However, SOAD must be predominantly process, rather than use-case driven. Therefore, a strong link between BPM and the use case modeling activities is required for SOAD.

On the design level, the goal of OO is to enable rapid and efficient design, development, and execution of applications that are flexible and extensible. Objects are software constructs that behave like the real-world entities that they model. For example, a Customer object will have a name, contact information, and might have one or more account objects associated with it. From an OO perspective, everything is an object.

The fundamental principles of OO are:

- *Encapsulation*: A software object is a discrete package containing both the physical properties (data) and the functionality (behavior) that mimics its real-world counterpart. An Account object, for example, maintains a balance and contains a mechanism for crediting to/debiting from that balance.
- *Information hiding*: Well-structured objects have simple interfaces and do not reveal any of their internal mechanisms to the outside world. A real-world example for information hiding is that you do not need to understand in detail how an automobile works to be able to drive it.
- *Classes and instances*: *Classes* are templates that define what kind of properties and behavior a given type of software object has, and *instances* are the individual objects that have values for those properties. Creating a new instance of a class is called *instantiation*. In a biological analogy, a human being is a class. All human beings have properties like height, weight, hair and eye color, and so on. Each of us is an instance of the class, HumanBeing, with a specific value for height, weight, and so forth. The classes last forever, and the instances have a limited lifetime.
- *Associations and inheritance*: The ability to express associations between classes and objects is a key concept in OO; inheritance is a strong form of association used to express is-a-relationships: in the same way that biological species are structured into a hierarchy of Kingdom, Phylum, Class, Order, Family, Genus, Species, we frequently find natural hierarchies of software objects. For example, when creating a financial application, Entities, you would probably need to construct classes like `CheckingAccount`, `SavingsAccount`, and `LoanAccount`. If you look a little closer (see Figure 3), you see that these classes share a lot of properties such as having a balance, the ability to credit or debit accounts, and so forth.

Figure 3. UML example class inheritance



Rather than duplicate the code that defines and manages these properties, you can create a common Account parent class that has a cash balance, and can handle credit and debit transactions. All the other classes are a specialized form of this Account object. For example, a LoanAccount will have a negative balance between zero and some agreed upon maximum, and a SavingsAccount will have a positive balance and will exhibit the behavior of adding interest, and so on.

- *Messaging*: In order to perform any useful work, software objects need to be able to communicate with each other. They do this by sending messages to one another. For example, a transfer of $1000 from a checking account to a savings account would be accomplished by sending a *debit* message, with argument $1000, to the `CheckingAccount` instance and a corresponding *credit* message to a `SavingsAccount` instance. When an instance receives a message, it executes a corresponding function, called a *method*, which has the same name as the message.

- *Polymorphism*: This term describes the situation where two or more classes accept the same message, but implement it differently. For example, both a `FreeCheckingAccount` instance and a `CheckingAccount` instance would respond to a `debit($100)` message, but the `FreeCheckingAccount` balance would debit its account balance by exactly $100, while the `CheckingAccount` instance would debit its balance by $100 plus transaction fee.

OO supports the full lifecycle for application analysis, design, and development:

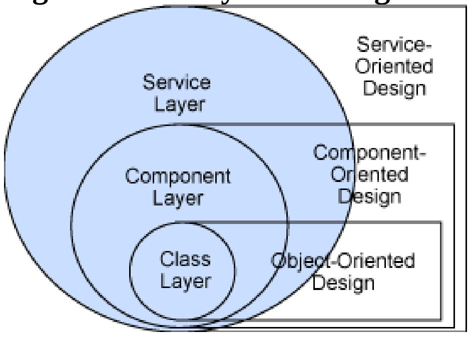
- *OOAD* attempts to find the optimal set of objects and the most natural class hierarchy to implement them.
- *OO development* focuses on incremental development of applications, implementing one business scenario or use case at a time. Tools like IBM WebSphere® Studio Application Developer help developers rapidly construct and test OO applications.
- *An OO runtime environment*, supplied by a Java™ Virtual Machine for example, executes code, provides application services like garbage collection (removal of resources that are no longer needed as the object that used them has been discarded), together with frameworks like J2EE that provide a mechanism for objects residing on different servers to intercommunicate.

The main issue with current OO design practices in relation to SO is that its level of granularity is focused at the class level, which resides at too low of a level of abstraction for business service modeling. Strong associations such as inheritance, create a rather tight coupling (and, consequently, a dependency) between the involved parties. In contrast, the SO paradigm attempts to promote flexibility and agility through loose coupling. There, currently, is no cross-platform inheritance support and first-class notion of a service instance in SOA in order to avoid having to deal with service lifecycle housekeeping issues such as remote garbage collection.

These considerations make OO difficult to align with the SO architectural style straightaway. However, OO still is a valuable approach for design of the underlying class and component structure within a defined service. Furthermore, many OOAD techniques such as classes, responsibilities, and collaborations (CRC) cards can be leveraged for service modeling, if elevated up to a higher level of abstraction. We will come back to this point later in this article.

Figure 4 illustrates the mapping between the levels of visibility and focus given by OO, *component-oriented*, and SO design. It also illustrates how they relate to each other within the context of SOA and SOAD.

Figure 4. The layers of design

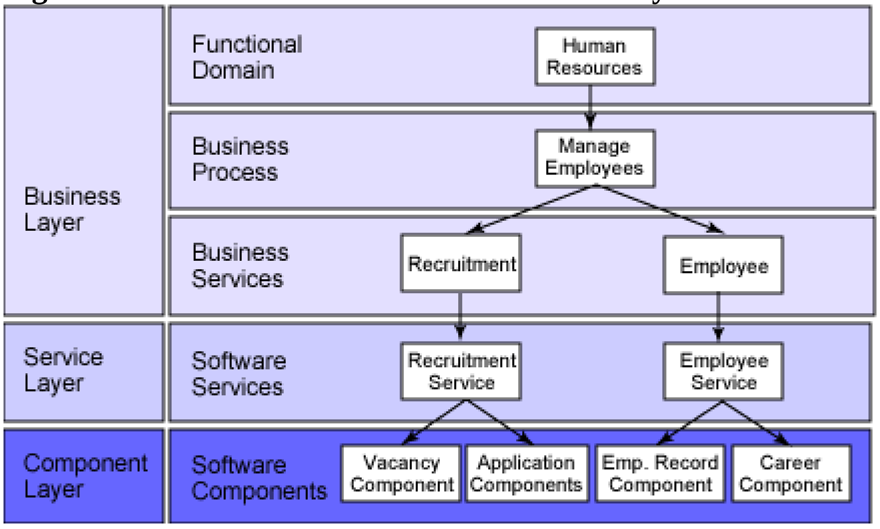


Regarding notation, the Unified Modeling Language (UML) -- enhanced with some additional stereotypes and profiles -- naturally becomes a key element of SOAD.

In the process area using the Rational Unified Process® (RUP) -- recognized as one of the leading OOAD processes to support the analysis and design of iterative software development -- utilizing a UML model is of primary value. However, RUP has the principles of

OOAD as its foundation, and therefore, does not lend itself easily to be aligned to SOA design. From the viewpoint of RUP, the architecture of a system is the structure of its major components interacting via defined interfaces. Furthermore, these components are composed of decreasingly smaller components down to a class-level of granularity. In contrast, the architecture of the system in a SOA generally comprises stateless, fully-encapsulated, and self-describing services satisfying a generic *business service* that is closely mapped to the BPM, as demonstrated in Figure 5.

Figure 5. The SOAD service definition hierarchy



These services might be composed of a number of collaborating or orchestrated services. This does not exclude the OO viewpoint adopted by RUP, but rather implements another layer of abstraction above it. This super-layer is to encapsulate components that are designed as RUP artifacts (as *software services*) within a formal, cross-layer interface structure.

SOAD elements

In this section, we look at the requirements for SOAD in more detail, and start to define its themes and elements. The objective is to ignite the discussion on this topic to further design work. Future work is certainly required to formalize the SOAD approach.

What does SOAD have to provide?

The following requirements have been identified for SOAD:

- Process and notation have to be formally (or at least semi-formally) defined, just like in any other project or design methodology. SOAD does not have to start from scratch. By selecting and combining OOAD, BPM, and EA elements, extra elements can then be defined if needed.
- There must be a structured way to conceptualize services:
  - OOAD gives us classes and objects on the application level, while BPM has event-driven process models. SOAD has the issue of gluing them together.
  - The method is no longer use case-oriented, but driven by business events and processes. Use case modeling comes in as a second step on a lower level.
  - The method includes syntax, semantics, and policies. This is required for ad hoc composition, semantic brokering, and runtime discovery.
- SOAD must provide well-defined, quality factors and best practices (such as answering the granularity question). The roles question raised by BPEL must be answered. For example, Who is responsible for which portion of the work: the Developer, Architect, or Analyst?

- The SOAD movement will also have to answer the question: What is not a good service? For example: anything that is not, or never likely to be reusable, probably does not make a good first-class SOA citizen (a service, that is). Another example is embedded, real-time systems with challenging, non-functional requirements, which cannot afford any XML processing overhead.
- SOAD must facilitate end-to-end modeling and have comprehensive tool support. If SOA is supposed to bring flexibility and agility to the business, the same should be expected from its supporting method, spawning from the business to the architecture and application design domains.

Quality factors

Some general principles or quality factors can already be identified and act as a design baseline within SOAD:

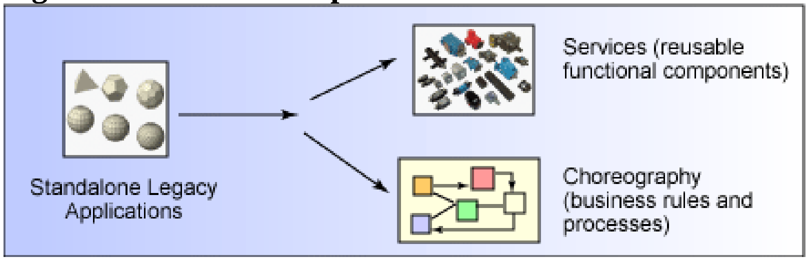
- Well-crafted services bring flexibility and agility to the business; they facilitate ease of reconfiguration and reuse through loose coupling, encapsulation, and information hiding.
- Well-designed services are meaningful and applicable for more than enterprise application; dependencies between services are minimized and explicitly stated.
- Services abstractions are cohesive, complete, and consistent; for example, one should think of the Create, Read, Update, Delete, and Search (CRUDS) metaphor when designing services and their operation signatures.
- A frequently stated assumption is that services are stateless (for example, not conversational); this statement would be weakened to request services to be as stateless as possible in the given problem domain and context.
- The service naming is understandable for domain experts without deep technical expertise.
- In a SOA, all services follow the same design philosophy (which is articulated through patterns and templates) and interaction patterns; the underlying architectural style can easily be identified (for example, during architecture reviews).
- The development of the services and service consumers requires only basic programming language skills in addition to domain knowledge; middleware expertise is only required for a few specialists, that in an ideal world, work for tool and runtime vendors, and not for the companies crafting enterprise applications as SOAs.

Service identification and definition

Top-down, business-level modeling techniques such as CBM can provide the starting point for the SOA modeling activities. But as already noted, a SOA implementation rarely starts on the green field; creating a SOA solution will almost always involve integrating existing legacy systems by decomposing them into services, operations, business processes, and business rules (also see Figure 6):

- Existing applications and vendor packages are factored into sets of discrete services that represent groups of related operations (bottom-up approach).
- Business processes and rules are abstracted from the applications into a separate BPM, managed by a business choreography model.

Figure 6. Service decomposition


Standalone Legacy Applications
Services (reusable functional components)
Choreography (business rules and processes)

All OOAD techniques can be applied in relationship to identifying and defining a service; however, a higher viewpoint needs to be taken. Furthermore, as SOAs aim higher than the classical development project, there is room for additional creative thinking.

Direct and indirect business analysis

BPM and direct requirements analysis through stakeholder interviews and CBM are an obvious and well-suited way of identifying candidate services.

Past experience shows that this main path should be amended by complementary, indirect techniques. When mining for candidate services, product managers and other business leaders should be interviewed. For example, what are the planned payment and billing models? The organizational chart of the enterprise that is supposed to use the system under construction should also be consulted. Any existing use case models from non-SOA projects should also be consulted for advice. The terminology used on marketing presentations for the system under construction is another great source of input regarding service operation candidates (the verbs in particular; marketing adverbs are not much of a concern!).

Domain decomposition

Domain decomposition, subsystem analysis, goal model creation, and related techniques, as defined in Endrei, are a first promising proposal for a SOA process structuring method or service conceptualization framework, which builds on earlier work by Levi and Arsanjani. The FODA work from SEI also contributes to this discussion.

Service granularity

To select the right level of abstraction is a key service modeling issue. You will frequently hear the advice to model coarse-grained. This is a slight oversimplification. You should rephrase it to model as coarse-grained as possible, without losing or compromising relevance, consistency, and completeness.

There is room for fine-grained service abstractions in any SOA, assuming that there is a business need. As SOA does not equal Web services and SOAP, different protocol bindings can be used to access services residing on different levels of abstraction. Another option is the bundling of several related services into coarser-grained service definitions, which is a variation of the facade pattern.

Naming conventions

An enterprise-wide naming scheme (XML namespaces, Java package names, Internet domains) should be defined. A simple example would be to recommend always assigning a service with a noun, and its operations with verbs. This best practice originates from the OOAD space.

First genuine SOAD elements

In addition to the combination of OOAD, BPM, and EA techniques, there are several important SOAD concepts and aspects, which have yet to be fleshed out:

- Service categorization and aggregation
- Policies and aspects
- Meet-in-the-middle processes
- Semantic brokering
- Service harvesting and knowledge brokering

Service categorization and aggregation

Services have different uses and purposes; for example, software services can be distinguished from business services (as shown earlier in Figure 5). Furthermore, atomic services can be orchestrated (composed) into higher level, full-fledged services.

Service composition is simplified by executable models (such as BPEL modeled ones); this is something traditional modeling tools and methods do not deal with.

Policies and aspects

A service has syntax, semantics, and QoS characteristics that all have to be modelled; formal interface contracts have to cover more than the Web Services Description Language (WSDL) does. Therefore, the WS-Policy framework is an important related specification.

Business traceability is a desirable quality, in addition to the well-established principle of architectural traceability: it should be possible to directly link all runtime artifacts directly to the language a non-technical domain expert can understand. This is particularly important for abstractions directly exposed at the business and service layer. The Sarbanes-Oxley (SOX) act is an example for a business driver requiring this kind of business traceability.

Process: meet-in-the-middle

There are no green field projects in the real-world as legacy applications (legacy application is a synonym for existing applications) always have to be taken into account. Therefore, a meet-in-the-middle approach is required, rather than pure, top-down or bottom-up process.

The bottom-up approach tends to lead to poor business-service abstractions in case the design is dictated by the existing IT environment, rather than existing and future business needs. On the other hand, top-down processing might cause insufficient, non-functional requirement characteristics, and compromise other architecture quality factors (for example, performance problems due to lack of normalization in the domain model) as well as provide impedance mismatches on the service and component layer.

Semantic brokering

In any SOA context, a formal interface contract for the invocation syntax is important. The semantics issue (the meaning of parameters and so forth) has to be solved as well (domain modeling). This is key in any business-to-business (B2B) and dynamic invocation scenario. Such scenarios are cornerstones of the SOA vision of being flexible and agile in response to the new business needs in a world of mergers and acquisitions, business transformation, globalization, and so forth.

Service harvesting and knowledge brokering

This is a knowledge management and lifecycle issue: how can services successfully be prepared and made available for reuse once they have been conceptualized?

Services should be identified and defined with reuse (and harvesting) as one of the main driving criteria of the SOA in mind. If a component (or service) has no potential for reuse, then it should probably not be deployed as a service. It can be connected to another service associated with the enterprise architecture, but will not be a service in its own right.

However, even if reuse is planned for right from the beginning, the process still has to formalize the service harvesting process. Service usage by more than one consumer is an explicit SOA design goal. A build-time service registry, for example, an enterprise-wide UDDI directory can be part of the answer.

Example: Automotive Work Order

The Automotive Work Order domain describes the process by which an automotive maintenance company manages its customer operations. We will use this domain problem to illustrate the issues of SOAD.

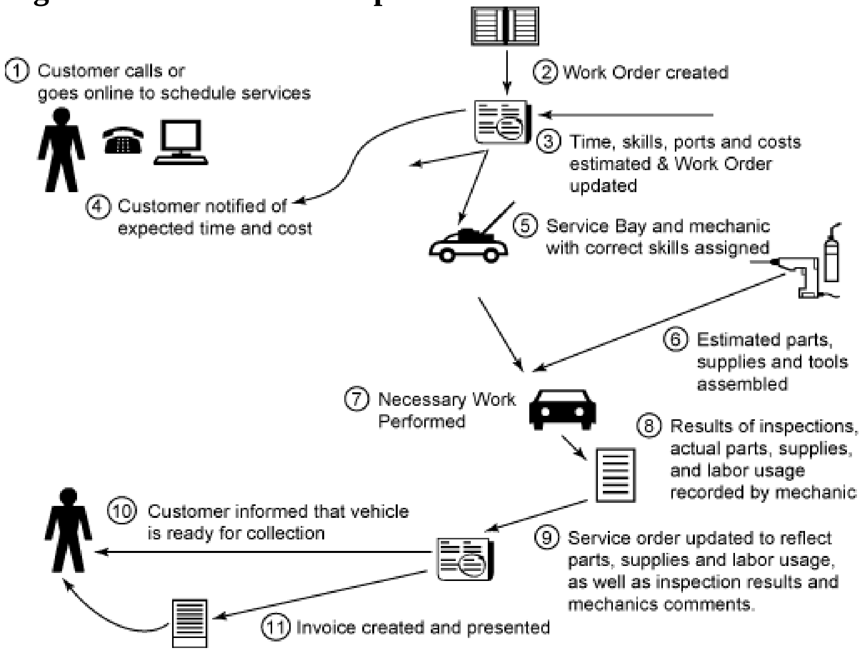
A work order represents an agreement between the auto service company and a customer to perform a set of scheduled or emergency maintenance activities such as a scheduled 50,000-mile service, a brake pad or tire replacement, or an oil change.

The business scenario (as shown in Figure 7) is as follows:

- The work order is created when the customer calls to make an appointment.
- For each planned maintenance activity or operation, a separate work order item is created, containing details of the expected usage of parts, supplies, and labor.
- The inventory is checked to ensure that all necessary parts are in stock before the appointment is scheduled.
- A suitably-equipped service bay plus a suitably-qualified mechanic needs to be scheduled for each work order item.
- The estimated total cost is calculated, and the customer approves the appointment, or the scenario terminates and the work order is cancelled.
- Immediately before the appointment, the necessary parts, supplies, tools, and equipment are assembled in the selected bay.

- When the customer arrives, the planned activities are performed, plus any other activities that become apparent when the vehicle is inspected.
- Actual values for parts and supplies used and labor are recorded.
- On completion of all maintenance, the total cost is calculated.
- An invoice is created and presented to the customer.

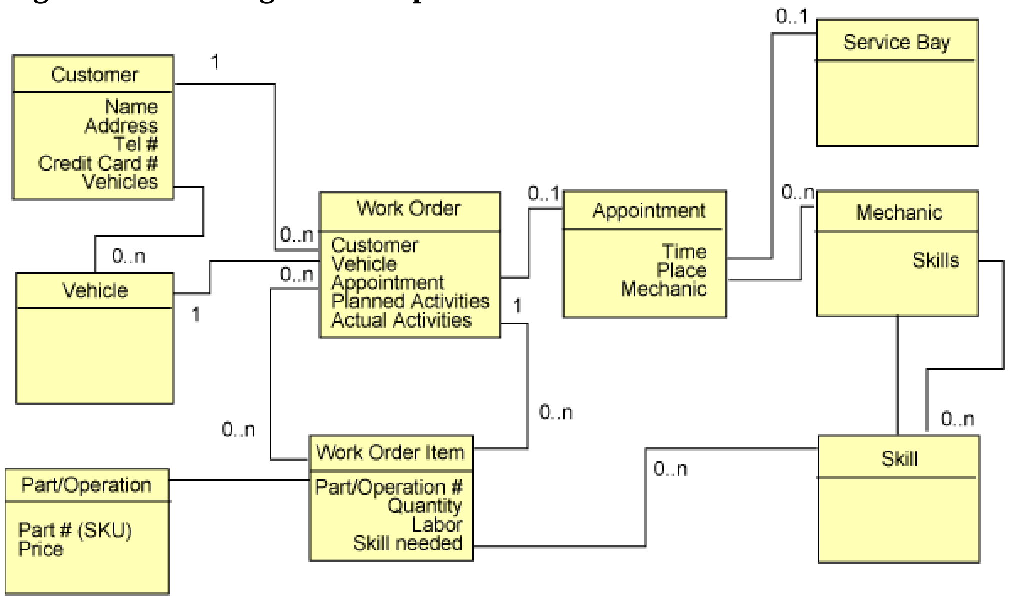
Figure 7. Macro flow example of Work Order



1 Customer calls or goes online to schedule services
2 Work Order created
3 Time, skills, parts and costs estimated & Work Order updated
4 Customer notified of expected time and cost
5 Service Bay and mechanic with correct skills assigned
6 Estimated parts, supplies and tools assembled
7 Necessary Work Performed
8 Results of inspections, actual parts, supplies, and labor usage recorded by mechanic
9 Service order updated to reflect parts, supplies and labor usage, as well as inspection results and mechanics comments.
10 Customer informed that vehicle is ready for collection
11 Invoice created and presented

If you were creating this from scratch as a stand-alone application, you might create a set of classes like those shown in Figure 8.

Figure 8. Class diagram example of Work Order



If you constructed Work Order as an OO application, these software objects would contain all the necessary business rules and would understand the business processes that should be followed.

However, there are some practical disadvantages to this approach:

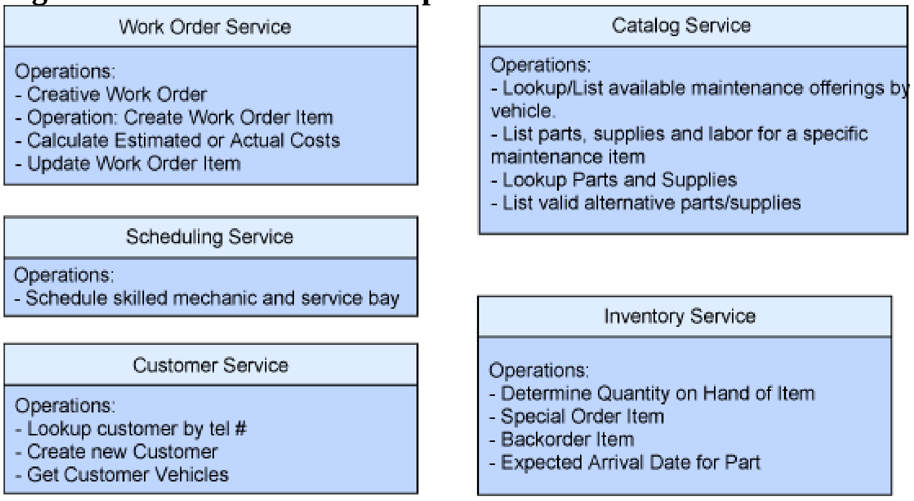
- Many of the steps involve interfacing with existing legacy systems and databases such as billing, scheduling, and inventory systems, which were probably not designed adhering to the OO paradigm (applying an adapter or mediator pattern helps in such situations).

- In order to make the system as flexible as possible, it would be helpful to have some of the rules externalized so that the processes or workflow can be modified without changes to the code. For example, rules like:
    - A standard 24,000-mile service includes four liters of oil. Additional charges for oil should only be made if more than four liters are used, or if the customer requests a premium grade oil (such as a synthetic oil).
    - There are a set of industry-standard labor estimates for common automotive maintenance activities available through a legacy application. The customer should be billed the standard labor costs unless the mechanic exceeds that estimate by more than X% and reports a valid reason for the difference.
    - The customer should be contacted for confirmation if the estimate is exceeded by more than Y%.

SOAD provides an excellent solution to these issues. Since it groups services on the basis of related behavior, rather than encapsulated (behavior plus data), the set of services will be subtly different from a business object model.

For example, you would probably group Work Order and Work Order Item together, into Work Order Services, and create Scheduling, Catalog, and Inventory services. Also, because there are no services instances, there are no equivalents to relationships between services. A Services Model for Work Order might look like Figure 9.

Figure 9. Services Model example of a Work Order


Work Order Service
Operations:
- Creative Work Order
- Operation: Create Work Order Item
- Calculate Estimated or Actual Costs
- Update Work Order Item
Catalog Service
Operations:
- Lookup/List available maintenance offerings by vehicle.
- List parts, supplies and labor for a specific maintenance item
- Lookup Parts and Supplies
- List valid alternative parts/supplies
Scheduling Service
Operations:
- Schedule skilled mechanic and service bay
Inventory Service
Operations:
- Determine Quantity on Hand of Item
- Special Order Item
- Backorder Item
- Expected Arrival Date for Part
Customer Service
Operations:
- Lookup customer by tel #
- Create new Customer
- Get Customer Vehicles

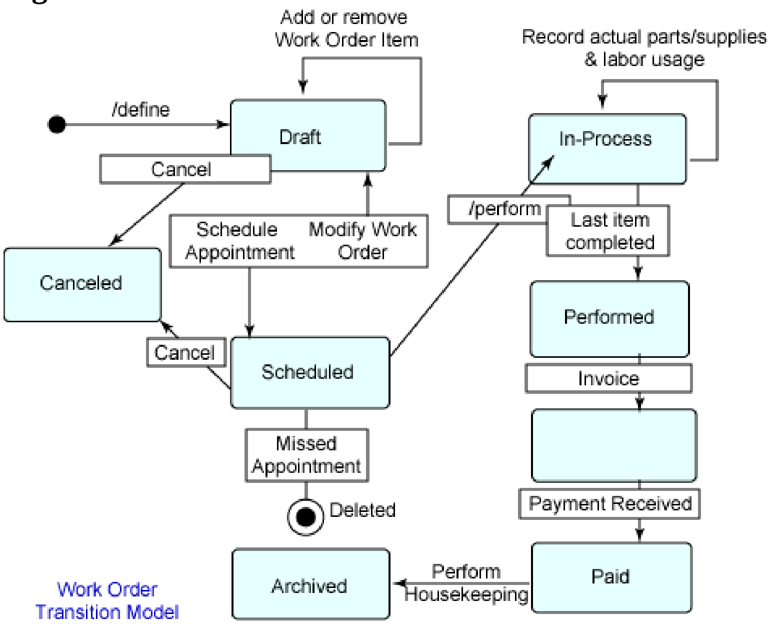
Unlike the OO paradigm, this model does not represent a functional system. There is no sense of flow, nor description of business events or rules. In the SOA paradigm, business process choreography, maintained externally to the services, determines the sequence and timing of execution of the service invocations.

Conceptually, the entire business from first customer contact, through completion of the work and payment of the bill, represents a single, macro-level unit of work, with a lifetime of days to weeks. After all, that unit of work generates revenue from a business perspective.

However, what occurs in practice is a series of intensive activities that take place (for example, defining the activities, scheduling an appointment, selecting parts and supplies, and performing maintenance activities) separated by relatively long periods of inactivity. In the IT system, there is no real process that lasts more than a few minutes; the state of the business process persists as data in a database between events.

This type of process can be well represented in state transition models (as, for example, available in UML). Figure 10 shows an example of how business flow can be modeled using a finite-state machine approach. It is a high-level view of how the state of a Work Order can change, as the business process progresses.

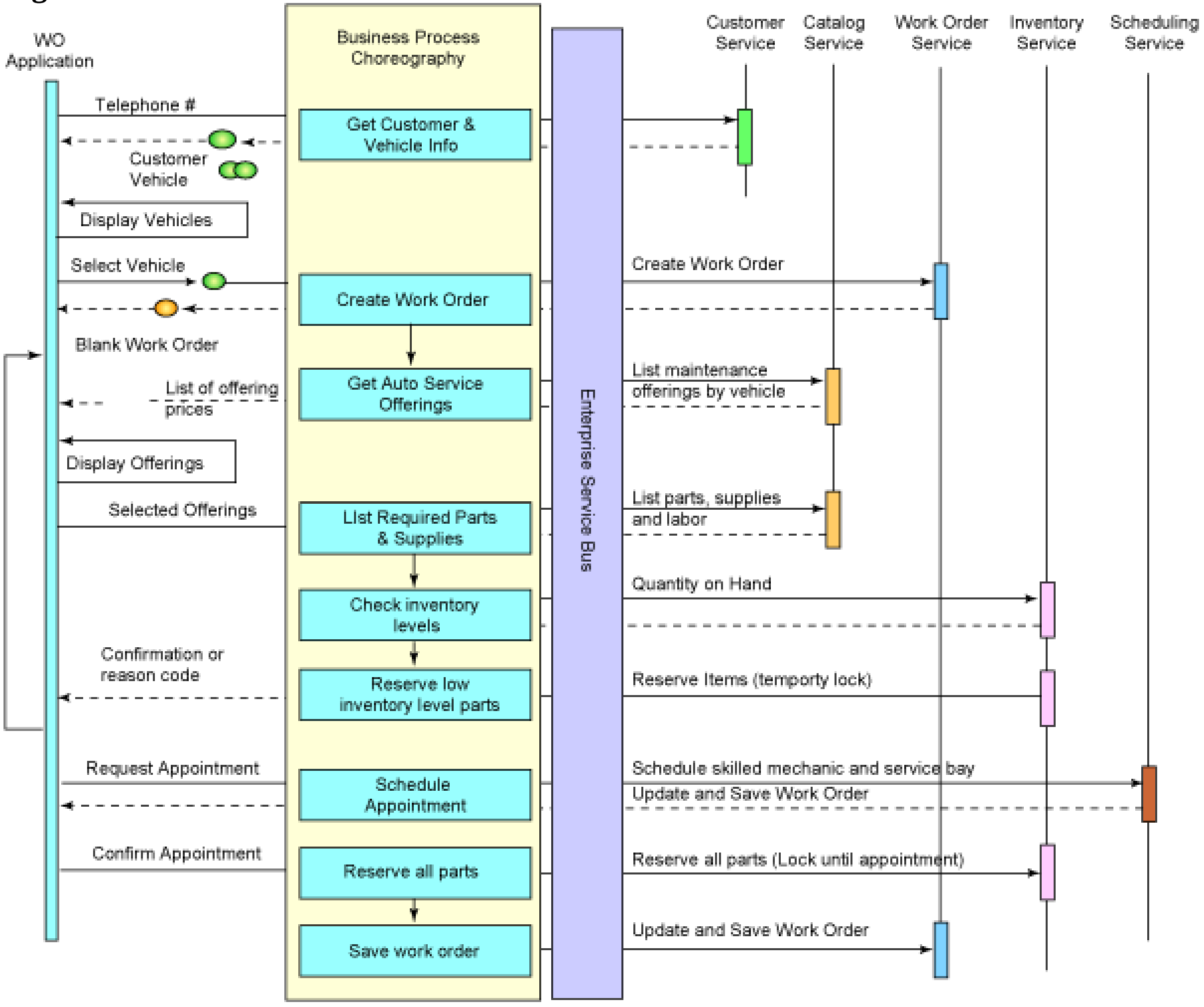
Figure 10. BPM for Work Order



Business process choreography focuses on these transitions between states. Individual operations record the associated state changes permanently.

Figure 11 shows an example of a partial choreography, covering steps 1 through 5 of the business scenario shown in Figure 10 (for example, the customer defines the work needed on their vehicle, and schedules an appointment to have the work carried out).

Figure 11. Business Interaction Model for Work Order



Conclusions and outlook

In this article, we have discussed and motivated the need for an innovative, meet-in-the-middle approach, bridging the business to IT gap, and supporting the analysis and design phase of SOA projects. We have proposed that this new, interdisciplinary SOAD approach is to be a holistic, modeling discipline that builds upon the well-established and proven OOAD, EA, and BPM foundations of today.

While SOAD notation and process remain to be defined in detail, key elements such as service conceptualization (or identification), service categorization and aggregation, policies and aspects, meet-in-the-middle process, semantic brokering, and service harvesting (for reuse) can already be identified.

SOAD will require enhancements to existing software engineering methods, further improving their usability and applicability to enterprise application development projects. Related best practices and patterns will evolve over time.

It is recognized that UML will continue to dominate as the notation of choice on the process side; enhancements will likely be required to satisfy the wider scope of SOAD.

The next steps on the road to a complete SOAD method will be to define the required end-to-end process and notation, review the required roles on engagements and their responsibilities, and continue to validate the proposed approach on projects.