Service Design Process for Reusable Services: Financial Services Case Study

Abdelkarim Erradi[1,3], Naveen Kulkarni[2], and Piyush Maheshwari[3]

[1] School of Computer Sc. and Eng. University of New South Wales, Sydney, Australia
[2]SetLabs Infosys Technologies Ltd, Bangalore, India
[3]IBM India Research Lab (IRL), New Delhi, India
aerradi@cse.unsw.edu.au, Naveen_Kulkarni@infosys.com,
pimahesh@in.ibm.com

Abstract. Service-oriented Architecture (SOA) is an approach for building distributed systems that deliver application functionality as a set of business-aligned services with well-defined and discoverable contracts. This paper presents typical a service design process along with a set of service design principles and guidelines for systematically identifying services, designing them and deciding the service granularity and layering. The advocated principles stem from our experiences in designing services for a realistic Securities Trading application. Best practices and lessons learned during this exercise are also discussed.

1 Introduction

Service Oriented Architecture (SOA) is a promising architectural approach to integrate heterogeneous and autonomous software systems. It promises effective business-IT alignment, improved business agility and reduced integration costs through increased interoperability and reuse of shared business services. SOA decomposes a system in terms of loosely coupled and replaceable services that interact via the exchange of messages conforming to well defined contracts [5]. SOA principles place a strong emphasis on decoupling the service consumers from the service providers via: (1) strict separation of service interface description, implementation and binding, thus allowing service changes to occur without impact on service users (2) declarative constraints and policies to govern the service behavior and the interactions between collaborating services (3) message-centric and standards-based interactions between participating services, thus allowing easier interoperability between systems inside and across enterprise boundaries. The perceived value of SOA is that it provides a flexible model that allows new applications/services to be created through the assembly of existing internal/third party services. Additionally, some of the new business requirements can be realized by re-composition of component services rather than by changing the services implementation. Therefore, SOA can help reduce the integration costs via eliminating the redundancy of overlapping and duplicate functionality as well as the consolidation and reuse of services across processes, lines of business, or the enterprise.

Technology and standards are important in building service-oriented distributed applications but they are not sufficient on their own. Moving to service-orientation is a non trivial one and requires far more than simply wrapping software entities with Web services interfaces. An effective approach for modeling and designing services is crucial for achieving the full benefits of SOA. In this paper, we present the set of design principles and processes for identifying, designing and layering services in a repeatable and non-arbitrary fashion. These have been derived from an elaborate SOA example involving the modeling of financial services for Securities Trading domain. The rationale behind design decisions is captured and the lessons learned are reported.

The rest of the paper is organized as follows. In Section 2 we provide an overview of the securities domain focusing on the pain points inherent in this area. Subsequently, in Section 3 we briefly discuss our suggested service-based decomposition framework. Section 4 details the suggested service design for our case study. Section 5 presents the lessons learned and the key service design considerations. The last section concludes the paper and provides some directions for future work.

2 Background and Problem Area

Despite the wide range of advocated advantages associated with the introduction of SOA, comprehensive SOA implementation case studies continue to be scarce in the literature. This paper aims to present a practical service design process along with key design principles derived from a Stock Trading service enablement case study.

For our case study, the key issues that SOA adoption aims to address are: (i) Heterogeneous IT portfolio with proprietary and brittle point to point connections that impact flexibility, (ii) Redundant and overlapping functionality leading to cost overheads and increased time to market. A specific example may be the use of individual pricing engines along with individual market data servers for multiple trading instruments, (iii) Inflexible and costly legacy applications portfolio.

The main business drivers for adopting service-orientation for our case-study are: (i) accelerate the securities trade processing towards Straight Through Processing (STP) allowing the final settlement to happen on the day of transaction, (ii) Make the securities trading accessible from various channels such the Web and mobile devices.

Many researches from academia and industry are suggesting various approaches to guide the service modeling and design. One of the outstanding efforts is this space is IBM's Service-Oriented Modeling and Architecture (SOMA) [1]. SOMA is a methodology for the identification, modeling and design of services that leverages existing systems. It consists of three steps: identification, specification and realization of services. However, SOMA lacks openly available detailed description of the methodology, which makes it difficult to further analyze its capabilities.

3 Service Oriented Decomposition Process

Service-based decomposition is an iterative process for arriving at an optimal partitioning of business capabilities into services. The first step is to first establish clear and well-defined boundaries between collaborating systems, followed by reduction of interdependencies and limiting of interactions to well-defined points. The key tasks in the service oriented decomposition process include identification of services along with deciding service granularity and appropriate layering of services.

3.1 Service Identification

As shown in Figure 1, for service identification we advocate a hybrid approach combining top-down domain decomposition along with bottom-up application portfolio analysis. This yields a list of candidate services that further need to be rationalized and consolidated. The top-down analysis of a business may be decomposed into products, channels, business processes, business activities and use cases. The business activities are often good candidates for business services. For example, the activity of obtaining a price for a specific security during an equity trading business process may be a logical candidate service. On the other hand, a broker could offer equity trading as a product which requires instantiating order placement and settlement processes, whose activities could be realized by services harvested from functionalities embedded in existing applications. The harvesting can be facilitated by reverse-engineering techniques and tools to extract data and control flow graphs that provide different views of abstraction of operational systems.

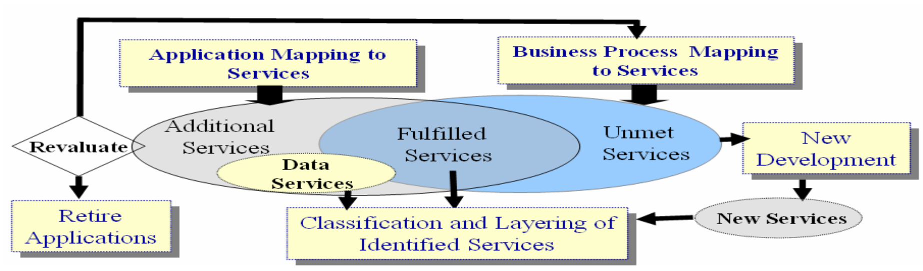

Fig. 1. Service identification framework

Our proposed service identification framework is initiated by a top-down capture and comprehension of key business processes as well as the mapping of the business processes to the existing application portfolio. This is followed by defining the To-be business process models (BPM) so that business services can be properly identified. BPM consists of the decomposition of the business domain into functional areas and business use cases. The level of functional decomposition of business processes depends on the level of complexity, for example a business process could be decomposed into business sub-processes, which in turn are decomposed into high-level business use cases comprised of a set of activities. For instance, the registration of a new customer is a business use case in a Trade Order process. The coarse grained business services are then defined on the basis of logical business activities. It needs to be noted that the services identified here may be applicable across use cases and business processes. Once the To-be BPMs are captured, a Process-to-Application Mapping (PAM) is required to examine existing software assets in order to discover candidate application functionality (e.g., APIs, sub-systems and modules from legacy, custom and packaged applications) for realizing identified business services. The mapping is performed between the business activities and the operational applications. This provides the basis for identifying applications that support a particular business process. Also the PAM helps to highlight possible redundancies and overlaps in the application portfolio, and to identify applications that offer potential shared services across channels and LOBs. In addition gaps and services that need new development can be uncovered. The important aspect of this exercise is that we end up with a conceptual map of the business services and maintain the association with the systems that may fulfill those services based on the existing IT portfolio. This is an important artifact that is essential towards matching the required services with existing services and to plan for new services that need to be built or acquired.

Apart from top down modeling, our framework also identifies functionality existing in the current enterprise IT portfolio. This can be accomplished by a combination of tools as well as interviews with application stakeholders. The outputs of this activity are typically fine-grained functional modules such as: updating customer's personal information, updating a customer's financial information, updating accounting entries for a cash payment transaction, etc. Collating all these functional activities will provide a comprehensive list of all the fine-grained activities performed by the application portfolio. This list of functionalities must be consolidated in a meaningful way to come up with reasonably coarse level activities that may be used to align with the services identified from the top down business process modeling effort.

The service identification also covers identifying reusable infrastructure services, currently supporting non-service oriented applications, which may be leveraged to support business services. For example security services providing authentication, authorization and secure communication, message delivery services to send messages and alerts to a variety of devices, such as email, SMS and fax. Another example might be provisioning services that manage subscriptions, SLAs, provisioning contracts, monitoring, metering and billing.

Figure 2 shows the meta-model we defined to guide service based decomposition activities. First the identification of candidate services starts with the services representing communication points between the parties involved. This is followed by capturing and describing the externally observable behavior of the identified services. In the current case study, the meta-model shown in Figure 2, provided the framework to identify the different types of services and their granularity.

An illustration of service-based decomposition of the Securities Trading application is depicted in Figure 3. During the service identification the primary view point should be towards achieving a common business goal through a single service. The business processes usually are modeled to achieve a single goal and hence would provide a natural boundary. For example a Trade Settlement service would aggregate various correlated activities like allocation matching, trade billing (commission, tax, fees etc) to achieve the goal of trade settlement.

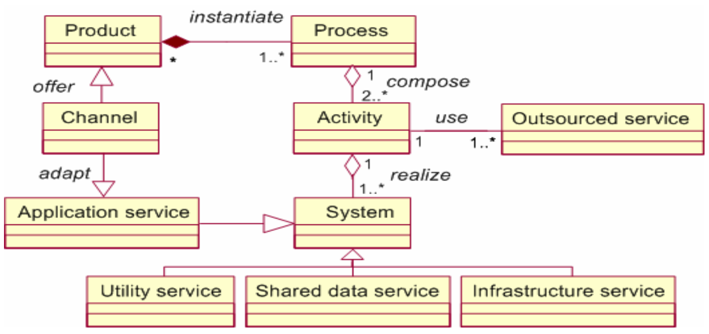

Fig. 2. Service conceptualization Meta-model

The identified services can be classified and grouped in a variety of ways. The services can be classified according to their scope into cross-business services, cross Line of Business (LOBs)/channels services, and LOB/channel specific services. The classification can also be based on their degree of reuse such as core enterprise services used by all (like a Customer Information Service), common services, or services unique to a specific application. The service classification activity is crucial to guide the non-functional aspects of services design, for example core and common services need to be designed and deployed with more emphasis on scalability and high availability.

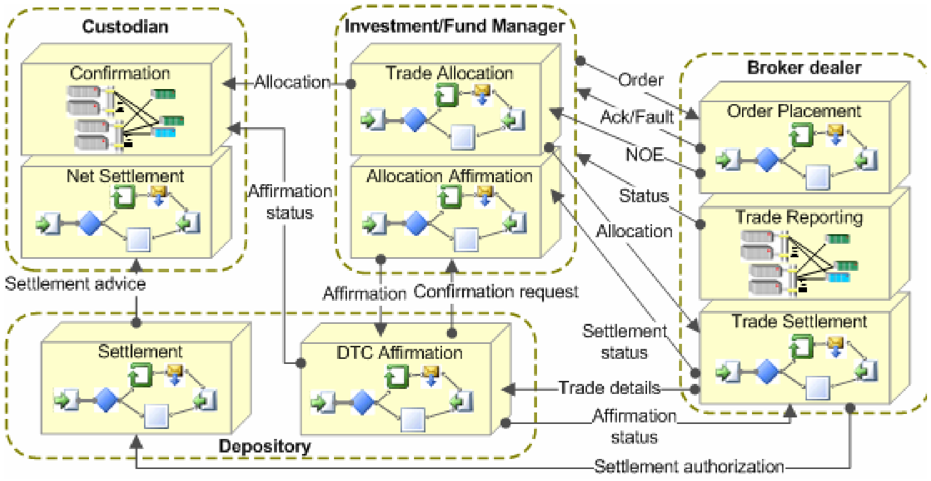

Fig. 3. High-level view of key Securities Trading services and their choreography

3.2 Service Granularity

The service granularity is considered a key design decision for service enablement. Services may be offered at different layers with different granularity. Service granularity refers to the service size and the scope of functionality a service exposes.

The service granularity can be quantified as a combination of the number of components/services composed through a given operation on a service interface as well as the number of resources' state changes like the number of database tables updated. The service should have the right granularity to accomplish a business unit of work in a single interaction. If the service is too coarse-grain, the size of exchanged messages grows and sometimes might carry more data than needed. Also it yields more complex interfaces and represents more possibilities to be affected by change. On the other hand if the service is too fine grained multiple round trips to the service may be required to get all the required data/functionality.

Hence a balance is struck, depending upon the level of abstraction, likelihood of change, complexity of the service, and the desired level of cohesion and coupling. A tradeoff needs to be made while taking into account non-functional requirements particularly performance.

Deciding the appropriate service granularity remains a challenge, but generally speaking services exposed to other systems should provide operations that correspond to business functions and they should be sufficiently generic to allow their reuse in different processes and/or by different users. Fine-grained component services may be used within a business service, but should not be exposed to other systems.

We have employed a business driven approach guided by the meta-model presented earlier to arrive to pragmatic granularity. The identified services, such as Trade Order Service and Trade Settlement Service, are business meaningful services that offer a single operation to fulfill a complete business task. Notice that we refer to the services as nouns, not verbs. In the contrary, focusing on the actions (verbs) rather than the service (nouns), such as Add Trade Order, often yields fine grained services.

There is no theory-founded method for deciding the correct level of granularity. The following guidelines can help in defining an acceptable level of granularity:

- **Reusability:** the optimal service design with respect to reusability is to provide a generalized set of services, compared to the development of a specific service for a specific consumer application. This enables the users to assemble a wide array of business applications using these services. Increased reusability stems mainly from accurate, complete and generalized service contract design capturing all possible message variants. This allows covering a larger number of usage scenarios through altering the service behavior simply by supplying varying message instances conforming to a subset of a super-schema defined by the service contract. For example designing an Insurance Quote Service based on a comprehensive schema definition like ACORD [2] allows the service to serve Quotes for individual as well as corporate users regarding various life insurance products and their variants. In the current case study, the process services such as Order Placement Service or Trade Settlement Service were envisioned to be reused across various products.

- **Business-alignment:** exposed business services need to add tangible business value and support a business use case. A service could be designed to represent a single important business concept, like a customer information service, thus forming clear traceability to the business model.

- **Design for assembly:** it is important that a service interface is defined in a way that its encapsulated functionality can be used and composed in different contexts with minimal effort so as to increase the service reuse potential. Simply exposing services directly off existing systems often yields non-optimal services that require considerable effort by the consumer to aggregate and refine them into useful services. Also the service interface should not be unnecessarily complicated so that it can be used and assembled with little complexity.

- **Reduce ripple-effects of applications changes:** services need to be self-contained and encapsulated in a way so that changes behind the interface can be done with no or minimal disruption to the service consumers. This increased isolation helps reduce change propagation and contain regression testing efforts and in turn reduces maintenance and evolution costs. In addition existing services may be swapped by new service implementations from potentially different providers without disturbing the service users.

- **Performance and size:** Services are often accessed remotely and might incur significant overhead to making a round trip. Hence the service design should expose coarse-grained operations covering a greater range of related functionality within a single service invocation in order to reduce the number of Service requests necessary to accomplish a task. In other words, a service should expose a significant business process capability, as opposed to low-level business functions. For example, the Trade Order Service should offer one operation (e.g., Place Order) to accept a Trade Order in one call instead of offering multiple operations consisting of "Create Trade Order Header" followed by a call to "Add Line Item" for each line item. However, coarse-grained operations might yield large size messages. Hence the size of messages should be constrained to what the service can process efficiently. So, the optimal size of exchange messages could guide the required adjustments to the service granularity.

4 Service Design

This Section briefly presents key service design principles. Then it discusses the main service design decisions for our Securities Trading case study and their rationale.

4.1 Service Design Principles

The service design should take into account the basic principle of high cohesion and low coupling among services [4] in order to minimize interdependencies and the impact of change while facilitating reuse. This ensures that the resulting services are self-contained, replaceable and reusable. **Service Cohesion** refers to the strength of functional/semantic relatedness of activities carried out by a service to realize a business transaction [4]. High cohesion ensures that a service represents a single abstraction and exposed interface elements are closely related to one another. **Service Coupling** refers to the extent to which a service is inter-related with other services, in other words it measures the degree of isolation of one service from changes that happen to another [3]. Low coupling can be achieved by reducing the number of connections between services, eliminating unnecessary relationships between them, and by reducing the dependencies between services to few, well-known dependencies [4]. Additionally, the service interfaces should be defined to be as independent as possible from the service implementation. This allows services to be independently deployed, and allows the assembly of applications that make no assumptions about service implementation beyond the characteristics published in the service contract. This way the service implementation can change without affecting service users so long as the service interface is unchanged.

Another key service design principle is that of stateless service design, services should not require context or state information of other services, nor should maintain state from one request to another. This implies that the exchanged messages should be self-contained with sufficient correlation information and metadata (such as links to persisted data) to allow the destination service to establish the message context [5]. On the contrary, a stateful interface tend to increase coupling between the service consumer and provider by associating a consumer with a particular provider instance.

Additionally, the service interface should be expressed in terms of meaningful business operations rather than generic or fine-grained primitive methods such as CRUD (Create, Read, Update and Delete) interfaces. The operations should correspond to specific business scenarios such as placing an order. Additionally the message contracts associated with the service operations should be coarse-grained encapsulation of business domain entities.

Sound interface design has to anticipate and meet the current and future needs of varied clients using the service in different contexts and different functional and QoS expectations. The service interface should capture and describe externally observable service behavior hiding the implementation details. This ensures that changes to the implementation are localized and do not necessitate changes in the service consumer.

The Service design should also accommodate multiple invocation patterns to be able to meet the requirement of various service consumers. A service consumer should be able to invoke the offered services using a variety of different invocation patterns such as synchronous invocation using SOAP over HTTP or asynchronous invocation using SOAP over JMS.

Optimal service granularity is crucial in ensuring maximum reuse in SOA. If the service is too coarse-grained, the size of the exchanged messages grows and sometimes might carry more data than needed. On the other hand if the service is too fine grained, multiple round trips to the service may be required to get the full functionality. Usually a balance is established, depending upon the level of abstraction, likelihood of change, complexity of the service, and the desired level of cohesion and coupling. A tradeoff needs to be made while taking into account non-functional requirements particularly performance. During service design, reusability can be maximized by using generalized service schema design, where the variations of the service behavior can be captured simply by supplying varying message instances conforming to a subset of a super-schema defined by the service schema.

4.2 Service Design Tasks

SOA is more about assembly of an integrated whole from independent parts. Hence, sound interface design is the essence of the integration design and it is a key tenet for reusable services. The challenge is that the service interface design has to anticipate and meet the current and future needs of varied clients using the service in different context and with different functional and QoS expectations. The service interface should capture and describe externally observable service behavior without leaking the details of the underlying implementation nor the service inner working and internal object model. Following this principle ensures that changes to the implementation are localized and minimize required interface changes.

Designing service-oriented applications involves a variety of tasks that may be enumerated as below, the aim to produce the design artifacts shown in Figure 4:

- Specifying the information model of the service as well as the structure and the data types of exchanged messages using a schema definition language such as XML Schema. The outcome of this task is to produce the Service Contract along with the associated Operations Contract, Messages Contract, Data and Faults Contract.
- Defining the behavioral model of the service comprising the service operations as well as the incoming and outgoing messages that are consumed or produced by the service. The service interface should also specify the supported Message Exchange Patterns (MEPs), such as one-way/notification and request-response pattern.
- Modeling of supported conversations and the temporal aspects of interacting with service, such as defining the order in which messages can be sent and received. For example, in the Order Placement Service, the actions available to a service consumer include presenting credentials, then placing an order.
- Specifying the service policy to advertise supported protocols, the constraints on the content of exchanged messages and QoS features, such as security, availability, response time, and manageability assertions. The key service attributes that need special attention are the transactional aspects of the service and whether the service is idempotent. These QoS requirements also dictate the Service Bindings and the Service Hosting options.

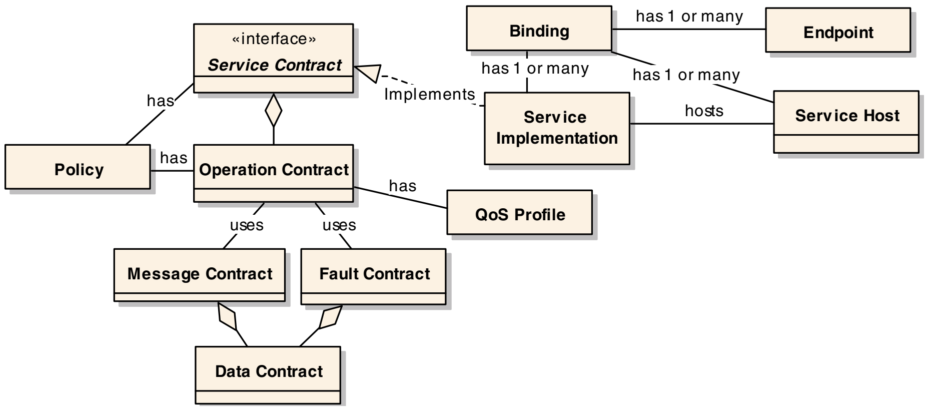

Fig. 4. Service Design Artifacts

4.3 Services for the Securities Case Study

The identified services are layered according to their granularity into four functional layers. Each layer has a set of roles and provides services to the layers above it. The top layer describes business processes made up of a sequence of business activities. The second layer defines business services that automate specific business process activities. The third level defines software components that allow the business services to leverage enterprise-level shared resources. The operational resources layer comprises applications, packages and databases that implement the services. For example, the Order Placement Service is implemented through wrapping relevant functionalities from the existing Order Management System (OMS) while the Allocation Matching Service is provided by the Trade Processing System (TPS).

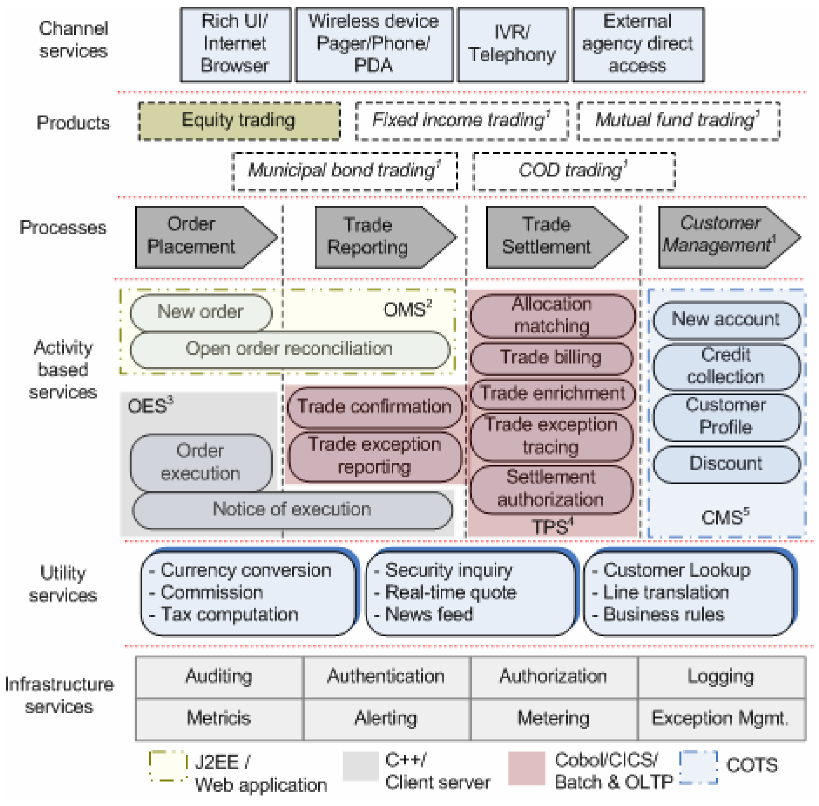

Fig. 5. Equity trading key services from the Broker viewpoint

Our design considers four types of services:

- Process services represent workflows that the Broker uses to deliver products offerings, like Equity Trading, through various channels like the Web, telephony or direct access. Process services, like Order Placement, expose access points that allow business partners to participate in the process. Process services also automate the information flow across disparate systems and eliminate duplicate data entry, manual data transfer and redundant data collection.

- Application services represent business activities that are useful across business units. For example, services like the Securities Pricing service is required across multiple business lines such as equity trading, fixed income trading, asset management, mutual fund trading etc. Application services provide shared and consolidated functional services to reduce/eliminate redundant/overlapping implementations.

- Shared data services map multiple schemas from different data sources to a single schema which is presented to collaborating applications. They provide the ability to unify and hide differences in the way key business entities are represented within the organization or between different business partners. Shared data services, like a

Customer service, can expose aggregated entities from specific data sources to reconcile inconsistent data representations and minimize the impact of change.

- Infrastructure services provide shared functions for other services, such as authentication, authorization, encryption, logging, etc. Often infrastructure services can be acquired, like an LDAP directory service, rather than built in-house.

5 Discussion and Lessons Learned

This Section discussed the key lessons learned from the Securities Trading case study. Further, key design considerations per service types are briefly presented.

5.1 Key Lessons Learned

While the SOA approach strongly reinforces well-established software design principles such as encapsulation, modularization, and separation of concerns, it also adds additional dimensions such as service choreography, scalable service mediation, and service governance. Our study highlights the following:

- Business process centered top-down identification of shared business services can lead to business aligned service design.
- An enterprise wide common information model (CIM), also known as Canonical Schema, is important to support the consistent representation of key business entities and to reduce syntactic and semantic mapping overheads between services. Standards like STPML [6] for the securities industry should be leveraged.
- Moving to SOA requires more than just a simple change of programming practices, rather a paradigm shift and mindset change is required to switch from RPC-based/object-based architecture to a loosely-coupled, message-focused and service-oriented architecture. A true SOA is realized when applications are built as self-contained, autonomous business services that interact by exchanging messages that adhere to specified contracts
- When service-enabling Mainframe CICS applications, it would be wise to expose one service per screen flow, and avoid translating all transactions to services. This involves identifying the required screens navigation to achieve key capabilities of the application, like CustomerCreation for instance, and then exposing the entire screen flow as a service.
- To ease service discovery and reuse, there is a need for clear service naming guidelines and a services metadata management repository to support governance and easy identification of services based on business function.

5.2 Design Considerations Per Service Type

For process services design the focus should be on the ease of modification and customization as these services are subject to higher change frequency. Hence, they should declaratively capture only the routing logic to manage the data and control flow between activity services. Further, complex business rules should be abstracted and externalized from processes so that they can be managed by a dedicated rules engine. Further, robust exception handling/compensation design is required.

Application services can have a verb-focused design by exposing key verbs as service methods, which unfortunately require RPC like behavior and sometimes might reveal the internal state of the service. We advocate a message-centric design to allow message content-driven service behavior and generalized service interface that can be used and composed in various applications. Command design pattern is used where the service performs dynamic content-based routing to direct the received messages to the appropriate implementation. This practice is acceptable when the resulting service contract is coherent and deals with closely-related business concepts. For example a generic Securities Price Lookup service could be provided to retrieve the price from various stock exchanges using content-based routing. Services need to be idempotent so that requests arriving multiple times are only processed once.

Shared data services uses noun-based design and usually expose CRUD interfaces representing simple atomic operations on an entity.

Infrastructure services are usually acquired and act on messages depending on the message context like the channel through which the message has arrived.

6 Conclusion and Future Work

Service-orientation is gaining momentum as a promising approach to deliver increased reusability, flexibility and responsiveness to change. However, the practical design of services requires sound engineering principles. The main contribution of this paper is a service-enablement case study in the securities trading domain to illustrate the issues and the challenges related to service design. The paper also emphasized the importance of service design in a Service-Oriented Architecture as well as the importance of focusing on the services' business value to guide the service requirement gathering, service identification and service design. Furthermore, we discussed the lessons learned with respect to the service design best practices and guidelines. Future work will focus on empirical studies of how the level of service granularity affects cohesion and coupling. We are also looking at developing an integrated toolset and a Domain Specific Language (DSL) supporting our service design methodology.

References

[1] Arsanjani, A.: Service-oriented modeling and architecture (SOMA) (2004), http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/

[2] Association for Cooperative Operations Research and Development (ACORD) (2007), http://www.acord.org

[3] Briand, L.C., Daly, J.W., Wüst, J.: A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering 25(1), 91–121 (1999)

[4] Papazoglou, M.P., van den Heuvel, W.J.: Service-Oriented Design and Development Methodology. Int'l Journal of Web Engin. and Technology (IJWET) (2006) (to appear)

[5] Parastatidis, S., Webber, J.: Realising Service Oriented Architectures Using Web Services. In: Service Oriented Computing, MIT Press, Cambridge (2005)

[6] Straight Through Processing Markup Language (STPML) (2007), http://www.stpml.org