Recipe for Service Component (R4SC)

Authors: Brian Paulsen, Buddy Ballentine, Jenny Ang

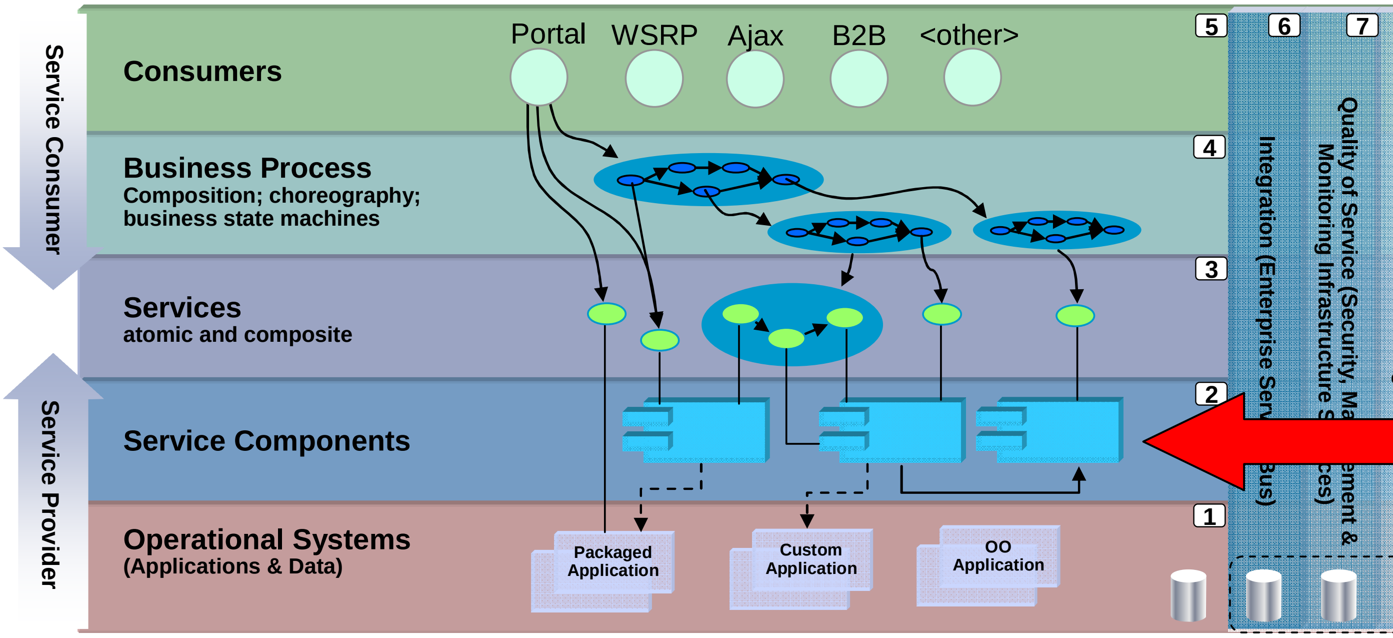
Key Contributors: Matthew Daflucas, Robert Warring, Abdul Allam, David Janson, Kishore Channabasavaiah, Siddharth N Purohit, Ali Arsanjani, Kerrie Holley

Vers 1.0
January 18, 2007

This paper is intended for architects and designers of SOA applications, focusing on the work they perform on the application architecture and design following the completion of the SOMA Business Analysis. The technique paper focuses on answering the question, “how one goes from the macro design of SOA solutions to their implementation”.

R4SC's Place in the SOA Landscape

The SOA Solution Stack consists of seven layers that make up the runtime environment of an SOA. The Recipe for Service Component focuses on addressing the entities that must work together to deliver the service component.



SOA Solution Stack

The Recipe for Service Component focuses on the components that implement the SOA Service on through to the integration with the operational systems, data sources, and other SOA services, themselves. In cases where you are integrating SOA services to make a larger grained SOA service, the components of the R4SC will also be acting as the Consumer. This document assumes you are familiar with the SOA Solution Stack, as well as SOMA. The intended audience is the developer community (e.g., Specialists, Architects, and others) involved in the design and implementation of SOA's.

This document is broken up into two sections, patterns and engagement scenarios. The engagement scenarios focus on the application of patterns in the delivery of our most common application situations, custom application development and integration development. During the discussion of the engagement scenarios, we will revisit the SOA Solution Stack (S3) and how the components map to the S3.

R4SC Design Patterns

- Service Component Pattern
- Integration Pattern

Service Component Pattern

The patterns and frameworks of object oriented best practices capture the best practices for particular design challenges found within application architecture. Acquiring the expertise to properly apply these patterns and frameworks, however, has proven to be a lengthy, expensive investment. The R4SC Service Component Pattern defines the best practices of integrating these patterns and frameworks to deliver an end-to-end architectural template for software engineering that helps accelerate the education of architects and developers on the application of OO best practices and helps organizations avoid the expense associated with trial-and-error application of the patterns and frameworks.

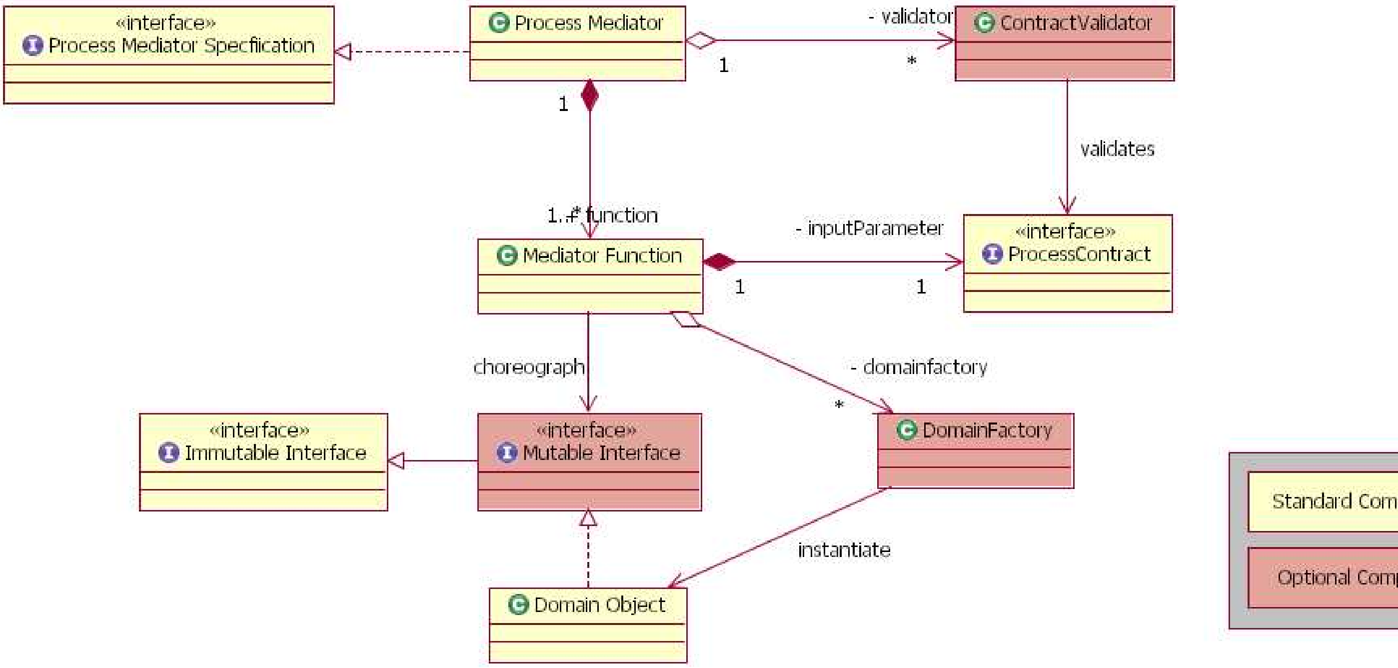


Figure 1 -- R4SC Service Component Pattern

Throughout this section we will reference a “view”, which is not to imply a user interface. Rather we are referring to View in the sense of the Model-View-Controller framework. In the case of an SOA, the view is the intergrator of the SOA service.

Process Contract

Responsibilty:

Process Contract consists of three elements: the process name, the process input, and the process output. Access to Process Mediators is controlled through a Factory/Command Pattern. The caller of a process must call the command function on the Factory, passing the Process Contract as its only input. In a pure SOA environment, this Factory will be an ESB's Registry. In a mixed SOA/non-SOA environment or in a pure non-SOA environment, the Factory will be a component that has the internal intelligence programmed to route SOA's to an ESB and non-SOA's to the appropriate application componentry.

Represents a list of requests for information that each Mediator Function (defined on the Process Mediator) will need answered in order to perform the requested operation. Process Contracts must be implementations of the Interface pattern and when defined represent the method signatures for the requests without their concrete implementations. Defining these objects as Interfaces is critical to afford the integrator the flexibility to implement according to its needs. When implemented, the Process Contract represents an independent object that encapsulates the answers to these requests and is passed to the appropriate Mediator Fucntion by the integrator.

Process Contracts enable the Process Mediator to "not care" which Provider Adapter is invoking its' Mediator Functions. In other words, they allow the Process Mediator to serve SOA-based requestors in the same manner it serves non-SOA requestors.

Architectural Decisions: (see full details of architectural decisions at Appendix: Architectural Decisions: R4SC Service Component Pattern)

- Process Contracts encapsulate the information needs of the Mediator Function and are abstract implementations of the Interface pattern.
- Process Contracts are only allowed to contain getter methods and inquiry methods (e.g., "is…")
- Process Contract methods can return native types, simple object types (e.g., String), and View Interfaces
- Process Contracts should be designed for use by a single Mediator Function.

Process Mediator and its' Mediator Function

Responsibilty:

The Process Mediator is the "brains of the operation". A Process Mediator should encapsulate **all** of the business process logic and rules. The Process Mediator is responsible for making Domain Objects, that are not (coupled) related, work together to provide a complete Mediator Function.

To accomplish this, the Process Mediator must work with the Domain Model, however, we do not want the Process Mediator to become tied directly to the current Domain Model, so we have the Process Mediator interact with Mutable Interfaces of the Domain Objects. There are three ways the Process Mediator will gain access to the Mutable Interfaces:

1. In the case that the Domain Objects must be retrieved from storage, the Process Mediator is responsible for triggering the retrieval of the Domain Objects and holding reference to the object through its' Mutable Interface. Triggering the retrieval of Domain Objects requires key information to find the desired object. The Process Mediator is responsible for requesting this information from the instance of the Process Contract it received and passing the information to the retrieval process.

2. In the case that the Domain Object must be instantiated as new, the Process Mediator is responsible for triggering the instantiation of the Domain Object and holding reference to the object through its' Mutable Interface. Typically, additional information will be supplied by the user to populate this new instance. The Process Mediator is responsible for requesting this information from the instance of the Process Contract it received and passing the information to the Mutable Interface for addition to the Domain Objects.
3. In the case that the Process Mediator requires a Domain Object previously retrieved/instantiated, the Process Mediator will request the Domain Object instance from the Process Contract instance that was passed to it as a parameter. The Process Contract will be able to return the Immutable Interface of that Domain Object instance, which the Process Mediator can cast – when needed – to its' Mutable Interface.

In all the case of option 1 or 2 above, Domain Object Factory may be leveraged to eliminate all references to the Domain Object class. See the section
Mutable Interface

Responsibility:
The Mutable Interface has two primary responsibilities:

1. Provide access to inquiry and modify the state of a Domain Object
2. Decouple the layers of the architecture from the Domain Object concrete implementation

The only users of the Mutable Interface are: Mediator Functions and other Domain Objects. Mutable Interfaces are implementations of the Interface design pattern and when defined represent the method signatures to inquire and change the public state of a concrete Domain Object implementation.

The Mutable Interface evolved and was added to The R4SC Service Component Pattern in order to decouple the Mediator Function from the Domain Model that it was originally built upon, thus allowing the Mediator Function to become a componentized asset that can be reused on top of other Domain Models.

Point of Interest:

> In order to integrate a Mediator Function on a different Domain Model, the new Domain Model must provide implementations for the Immutable Interfaces and Mutable Interfaces that the Mediator Function leverages.

The primary user of the Mutable Interface is the Mediator Function. The Mutable Interface defines a view on the Domain Objects (Model) that the Mediator Function (Controller) uses to inquire and modify the state of the Domain Model. Other users of the Mutable Interface include: Domain Objects, and other Mutable Interfaces. The Domain Objects use the Mutable Interfaces to decouple themselves from the specific implementations of other Domain Objects in the model. For all aggregate relationships between Domain Objects, the aggregate wholes should hold reference to their Domain Object aggregate parts through their associated Mutable Interface, rather than hold reference directly to the Domain Object. The aggregate whole Domain Object will modify the state of its aggregate part Domain Objects through the Mutable Interface, rather than directly messaging the Domain Object.

It is… a blueprint defining services to inquire and modify the state of a Domain Object

It is NOT… a specific implementation of the services defined, nor access to private or protected services of the Domain Object

Immutable Interface

Responsibilty:
In comparison to the Mutable Interface, an Immutable Interface provides reduced visibility to only inquire the state of its' Domain Object. The Immutable Interface's primary purpose is to preserve the Model-View-Controller framework by providing a solution that can be returned to the View, which prevents the View from having direct reference to the Model (Domain Objects), and prevents modifying the state of the Modek (Domain Objects). It has two primary responsibilities:

1. provide access to inquire (e.g., getter access) but not change (e.g., no setter access) the public internal state of a Domain Model object
2. decouple the layers of the architecture from a Domain Model object's concrete implementation

Immutable Interfaces are implementations of the Interface design pattern and represent the list of method signatures (without concrete implementations) that the user can request in order to receive information about the public state of a Domain Object, without becoming coupled to the concrete implementation of the Domain Object.

The primary user of the Immutable Interface is the Provider Adapter and its requestors. The Immutable Interface defines a view on the Domain Objects (Model) that can be returned by the Mediator Function (Controller) to the Provider Adapter (View) to preserve the integrity of the Model-View-Controller (MVC) framework.

Other users of the Immutable Interface include: Process Contracts, Mediator Functions, and Domain Objects. All of these objects should make inquiry requests to the Immutable Interface of a Domain Object, rather than talk directly to the Domain Object.

It is… a blueprint defining an abstract interface (e.g, API) with access to view the state of a Domain Object

It is NOT… a specific implementation of the services defined, nor does it provide access to modify the state of a Domain Object

Domain Factory for futher details.

Each Mediator Function on a Process Mediator must take a Process Contract as a parameter. Provider Adapters will only be able to access Mediator Functions by providing the appropriate Process Contract. The Mediator Fucntion will query the provided Process Contract for information it needs to complete the requested service, and will not forward the contract to objects out of the Controller Layer. The Mediator Function may use Mutable Interfaces and other Mediator Functions to deliver its' service. The Process Mediator is responsible for knowing the functions that need to be called and the order in which they must be called. If a Mutable Interface function needs to be called and parameters passed, the Mediator Function is responsible for either passing information it has gathered from services it has triggered or requesting information from the instance of the Process Contract that was passed to it. If another Mediator Function is to be called, the calling Mediator Function is responsible for instantiating the appropriate Process Contract and populating the contract.

The Mediator Fucntion is responsible for handling all exceptions that are returned from Mutable Interfaces, Contract Validator, and other Mediator Functions it leverages to deliver its' function. In the case that a recovery alternative does not exist for the caught exceptions, the Mediator Function is responsible for delivering business exceptions to the integrator that triggered it.

In order to eliminate data compatibility exceptions that may arise as a result of inappropriate input data into the Mediator Function process, the Mediator Function may choose to leverage A Contract Validator to confirm the state of the input data of the contract prior to using the data in it's processing. Appropriately written Process Contracts will eliminate most of the potential data input problems, thus allowing the Contract Validator to remain lightweight.

The Mediator Function rules focus on the business logic that pertains to the process and the interoperation of the unrelated Domain Objects. Domain logic that is specific to a particular Domain Object only should **NOT** be moved to the Process Mediator component.

Finally, Process Mediator's Mediator Funcstion are responsible for managing the logical units of work to successfully complete a business process. The Mediator Function should open, execute, and close the logical unit of work within the execution of its' function. Mediator Functions are stand alone operations, therefore, they do not have long running transactions. In the situation of a composite Mediator Function, a situation where a Mediator Function calls other Mediator Function(s) to fulfill its operation, the calling Mediator Function should be responsible for the logical unit of work. However, since Mediator Functions are expected to be able to stand alone, it is imperative that a Mediator Function be intelligent enough to determine if it is part of an existing logical unit of work, and if not how to open and control a new logical unit of work, itself.

Architectural Decisions:

- Process Mediators are intended to have many Mediator Functions
- Mediator Functions must take as their only parameter an abstract implementation of the Interface pattern : a Process Contract
- Mediator Functions can return one of five values: void, an Immutable Interface, a collection of Immutable Interfaces, a native type/Simple object (e.g., boolean or String), a collection of native types/Simple objects
- Mediator Function cannot pass reference to the Process Contract outside of the Process Mediator Layer
- Mediator Functions talk to the Domain Objects through the Mutable Interfaces
- A Mediator Function can leverage other Mediator Functions to delivery its service.
- A Mediator Function is responsible for managing the Logical Unit of Work

Contract Validator

Responsibilty:
This component is used to validate the state or content of a Process Contract for a Medatior Function. The contract must adhere to this validation or an exception will be raised. The Contract Validator is invoked by the Mediator Function, when needed/desired.

The purpose of the validation is to confirm that the minimum input data has been provided. The Contract Validator should not become a single source for all rules validation. Process oriented rules should remain in the Mediator Function, while rules internal to the Domain Object should remain within the Domain Object.

Another temptation is to propagate data and enterprise application validation checks to the Contract Validator. This cannot be allowed, as this will tie the Mediator Function logic of the application to the current back-end data sources and enterprise applications, making the solution less flexible to change. This typically becomes a temptation, because there is a desire to catch problems with the user provided information as soon as possible due to a perception that this will improve performance. It is critical to assess the non-functional requirements for performance to make sure that there is justifiable business reason to consider moving such validation to a higher layer in the application. In making the decision to move the validation up to a higher layer in the architecture, you are accepting a trade-off in maintainability and flexibility of the solution.

Our experience has shown that in most cases, the decision to duplication rules and move validation to a higher layer of the architecture was made before justifying through performance modeling. Our experience also has shown where the performance tests are run to confirm/disprove the need to move such rules up in the architecture, in most cases the move is not required in order to meet performance requirements. What should this tell you: that most application teams that decide to move the rules up in the architecture make the trade-off (which results in less flexible, less maintainable solutions) when it wasn't necessary to meet performance requirements.

The use of The Contract Validator has benefited projects that choose to use it by virtually eliminating null data exceptions in the solution.

Architectural Decisions:

- **Prohibit the duplication of rules in the Contract Validator.**
- Contact Validation should be managed by a support object that the Process Mediators leverage, rather than a function of the Process Mediators themselves.

Domain Object

Responsibilty:
Domain Objects are the "real world" entities that are used by Mediator Functions to complete complex processes. Domain Objects are responsible for encapsulating the rules that are pertinent to them internally, as well as its attributes.

Each Domain Object must have an Immutable Interface and Mutable Interface. The Domain Object is responsible for implementing the Mutable Interface, which in turn is responsible for extending the Immutable Interface. Aggregate whole Domain Objects should hold onto their aggregate part Domain Object references via each part's Mutable Interface.

Architectural Decisions:

- Aggregate Domain Objects hold reference to the Mutable Interface of each of its' parts
- Each Domain Object must implement it's corresponding Mutable Interface
- Instance creation and management of the Domain Object must be managed through a factory method.

Mutable Interface

Responsibility:
The Mutable Interface has two primary responsibilities:

3. Provide access to inquiry and modify the state of a Domain Object
4. Decouple the layers of the architecture from the Domain Object concrete implementation

The only users of the Mutable Interface are: Mediator Functions and other Domain Objects. Mutable Interfaces are implementations of the Interface design pattern and when defined represent the method signatures to inquire and change the public state of a concrete Domain Object implementation.

The Mutable Interface evolved and was added to The R4SC Service Component Pattern in order to decouple the Mediator Function from the Domain Model that it was originally built upon, thus allowing the Mediator Function to become a componentized asset that can be reused on top of other Domain Models.

Point of Interest:

In order to integrate a Mediator Function on a different Domain Model, the new Domain Model must provide implementations for the Immutable Interfaces and Mutable Interfaces that the Mediator Function leverages.

The primary user of the Mutable Interface is the Mediator Function. The Mutable Interface defines a view on the Domain Objects (Model) that the Mediator Function (Controller) uses to inquire and modify the state of the Domain Model. Other users of the Mutable Interface include: Domain Objects, and other Mutable Interfaces. The Domain Objects use the Mutable Interfaces to decouple themselves from the specific implementations of other Domain Objects in the model. For all aggregate relationships between Domain Objects, the aggregate wholes should hold reference to their Domain Object aggregate parts through their associated Mutable Interface, rather than hold reference directly to the Domain Object. The aggregate whole Domain Object will modify the state of its aggregate part Domain Objects through the Mutable Interface, rather than directly messaging the Domain Object.

It is… a blueprint defining services to inquire and modify the state of a Domain Object

It is NOT… a specific implementation of the services defined, nor access to private or protected services of the Domain Object

Architectural Decisions:

- Mutable Interface may provide access to public setter methods (service methods that change state) and public getter methods (to inquire the state of the object).
- Mutable Interface methods can return native types, simple object types (e.g., String), Mutable Interfaces, Immutable Interfaces, or void.

Immutable Interface

Responsibilty:
In comparison to the Mutable Interface, an Immutable Interface provides reduced visibility to only inquire the state of its' Domain Object. The Immutable Interface's primary purpose is to preserve the Model-View-Controller framework by providing a solution that can be returned to the View, which prevents the View from having direct reference to the Model (Domain Objects), and prevents modifying the state of the Modek (Domain Objects). It has two primary responsibilities:

5. provide access to inquire (e.g., getter access) but not change (e.g., no setter access) the public internal state of a Domain Model object
6. decouple the layers of the architecture from a Domain Model object's concrete implementation

Immutable Interfaces are implementations of the Interface design pattern and represent the list of method signatures (without concrete implementations) that the user can request in order to receive information about the public state of a Domain Object, without becoming coupled to the concrete implementation of the Domain Object.

The primary user of the Immutable Interface is the Provider Adapter and its requestors. The Immutable Interface defines a view on the Domain Objects (Model) that can be returned by the Mediator Function (Controller) to the Provider Adapter (View) to preserve the integrity of the Model-View-Controller (MVC) framework.

Other users of the Immutable Interface include: Process Contracts, Mediator Functions, and Domain Objects. All of these objects should make inquiry requests to the Immutable Interface of a Domain Object, rather than talk directly to the Domain Object.

It is… a blueprint defining an abstract interface (e.g, API) with access to view the state of a Domain Object

It is NOT… a specific implementation of the services defined, nor does it provide access to modify the state of a Domain Object

Architectural Decisions:

- Immutable Interface will adhere to the guidelines of the Interface pattern and be implemented by it's Business Object
- Immutable Interfaces are only allowed to contain getter methods
- Immutable Interface getter methods can return native types, simple object types (e.g., String), or Immutable Interfaces
- In the case that a Mediator Function is provided to initially retrieve a part of a Domain Object, the Domain Object's Immutable Interface should not expose the getter method for that part.
- Immutable Interface's getter methods should not have any parameters.

Domain Factory

Responsibility:
The Domain Factory is in place to provide complete encapsulation of the Process Mediator/Mediator Functions from knowledge of the concrete implementations of the Domain Objects. The Domain Factory is an implementation of the Factory design pattern, which is used by the Mediator Functions to request new instances of the Domain Objects. The Domain Factory is responsible for providing the requestor with an instance of the Domain Object, but preventing the requestor from directly accessing the instance of the Domain Object by returning only access to the Controller View. By

providing only access to the Controller View, the Mediator Function will remain independent of the implementation of the Domain Object, which will allow greater flexibility for the Domain Object to change without change to the Mediator Function. Only in the event that the public interface of the Controller View changes might the Mediator Function be affected.

The Factory pattern, though elegent in the decoupling it offers between the Process Mediator and the Domain Objects, introduces a concept called indirection. As indirection increases within a design and code it means that it is harder to trace how one component relates to another. Care should be taken in assessing the capability of the development and support team to manage the indirection introduced by the Factory Pattern. The purpose for using the factory is to allow for greater flexibility to change in order to reduce the cost of maintenance of the solution over time. For teams inexperienced in design patterns and their implementation, the indirection can increase the complexity of maintaining the solution. Before considering the Factory Pattern, the architect should assess whether the complexity introduced by the indirection of the Factory pattern will outweigh the benefits. Assuming the benefits outweigh the consequences of indirection, the following architectural decision should be followed:

Architectural Decisions: ▪ Both the Domain Factory and Mutable Interface must be applied to encapsulate the controller from the Domain model.

Service Integration Pattern

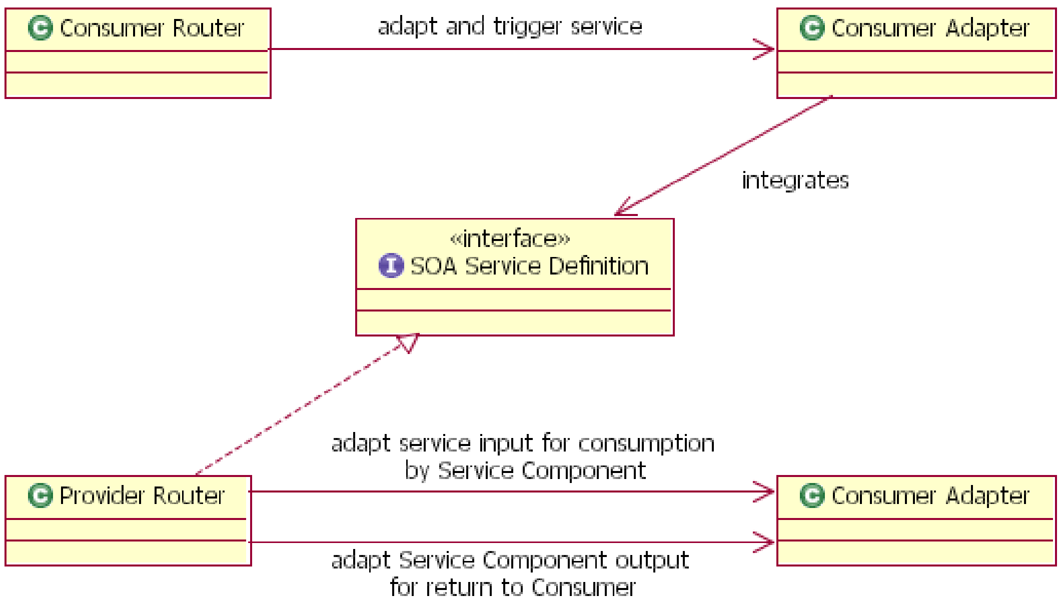


Figure 2 - R4SC Service Integration Pattern

We just described the pattern and responsibilities for defining the inner workings of a service component, now we must discuss how to take expose them as services and integrate those services into solutions. Many components within an application's layers may need to take advantage of SOA services. Though the requesting component will change, the pattern for integrating the service request should be consistent.

The principles of this pattern are to be applied between any consumer and provider of a SOA service. The purpose is to encapsulate both the consumer and provider business logic from the integration technology chosen to enable the SOA communication. On each side of the ESB is the Consumer Service Adapter and Provider Service Adapter, which act like a modem modulating and demodulating the information passed across the ESB from the transport data structure to a structure expected by the implementation language of the business applications on each end. Consider the Router to be responsible for the business aspects of the intergration and the Adapter responsible for the technical aspects of the integration.

Provider Router

The Provider Router is responsible for implementing the SOA service and routing the service request to the appropriate Business Mediator Function. The Provider Router is essential to keep the technology specifics isolated from the Business Mediator Function, thus allowing the business logic to be reusable by a wide variety of access channels leveraging different technologies. As the implementor of the SOA service, it is the Provider Router's responsibility to ensure the input data is translated into the format needed by the functional component. The Provider Router achieves this translation by calling a Provider Adapter and passing the input format to it. A populated instance of a Process Contract implementation will be returned by the Provider Adapter to the Provider Router, at which point the Provider Router will know which Process Mediator Function to call.

The Provider Router is also responsible for handling the response from the Process Mediator Function and repackaging for return to the service consumer. To do this, the Provider Router triggers an operation on it's Provider Adapter to translate the return information or error messages to the return format demanded by the SOA Service definition. Once the return data is properly formatted, the Provider Router triggers the proper response as dictated by the SOA Service definition and completes its execution.

The Provider Router has two primary responsibilities:

1. Trigger the appropriate translation for inbound and outbound data on it's Provider Adapter
2. Route the service request to the appropriate Process Mediator Function

Finally, the Provider Router fulfills the technical controller responsibilities (in model-view-controller terms) for accessing an SOA Service. Its' primary purpose is to implement the SOA service definition and ensure the proper process is triggered. The Provider Router should not have any process oriented logic; therefore, the Process Router is not allowed to control a logical unit of work. The reason for this decision is to prevent process logic from graduating out of the Process Mediator and into the Process Router over time.

Architectural Decisions:

- A Provider Router function is limited to only routing messages to a single Process Mediator Function to ensure process logic does not get distributed across a variety of design elements.

Provider Adapter

Responsibility:

Process Mediators, defined in the R4SC Service Component Pattern, are written with the intention of being used by a wide variety of access channels, which will vary in their technologies (e.g., programming language, communication protocol, data format, etc). So, we need a component that knows the format of the information that is coming in from the requestor and has the knowledge to translate that data to the format of the Process Mediator. That component is the Provider Adapter. The Process Contract is the format that must be translated into and expected by each Mediator Function that will be invoked by the Provider Router to fulfill the SOA service.

The Provider Adapter has two primary responsibilities:

- translate from the SOA service definition structure to the Process Contract required by the Process Mediator Function the Process Router will be triggering
- translate the result set returned to the Process Router at the completion of the called Process Mediator Function back into the response format defined on the SOA service definition

R4SC Engagement Scenarios

Custom Application Development Scenario

The custom application development scenario focuses on the application of patterns where a business model is deamed necessary within the new application development space. This scenario is typically presented when the existing operational systems functions do not provide out of the box functionality to address the SOA service needs of the organization. As a result, there is a need to integrate the existing operation system functions, and add some additional business and process logic in order to fulfill the demands of the business. As a result there is a three tiered service:

1. The Service definition and Adaptation layer (View Controller) – implementing the SOA service definition, translating to and from the technology format of the underlying application, and routing the service request to the application elements
2. The Business Mediator Layer (Business Controller and Business Model) – encapsulating the business processes and business entities of a Functional Component
3. The Integration Mediator Layer – encapsulating the data entities and technology integration to access the operational systems, data architecture, and integration technologies of the SOA Stack.

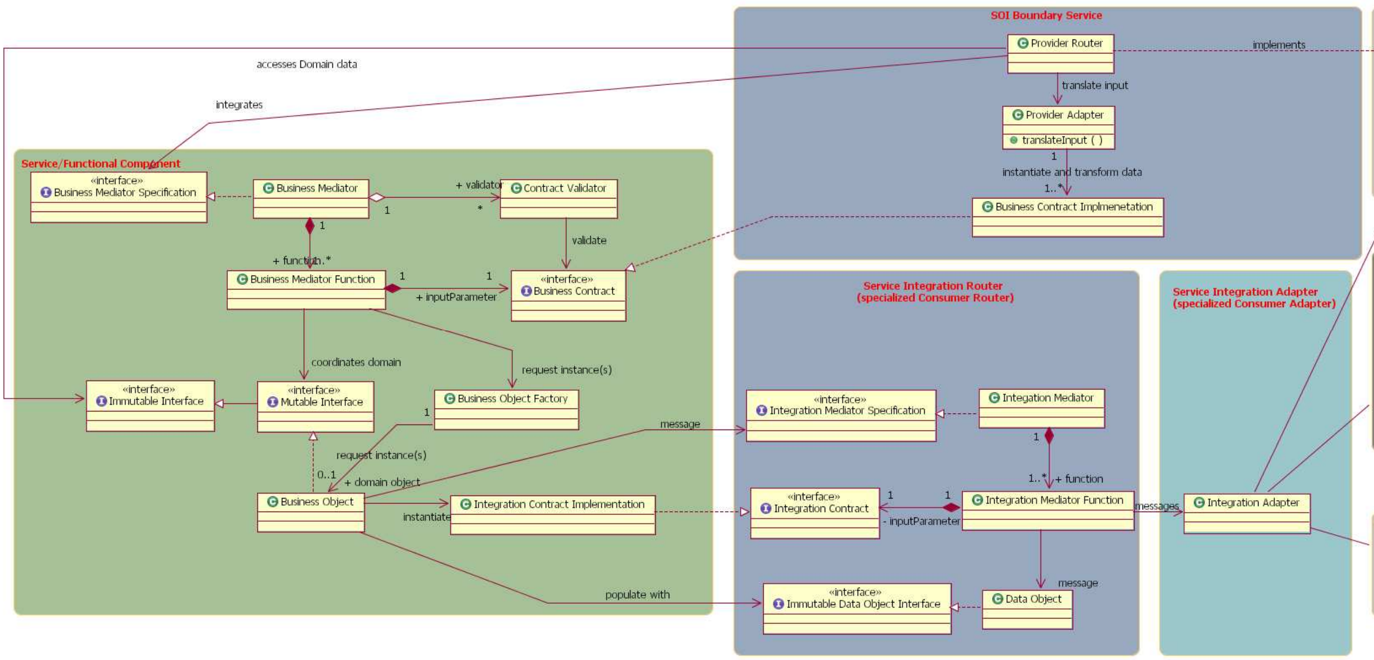


Figure 3 - R4SC, Custom Application Development Scenario

Controller Layer:

In the traditional Model-View-Controller, the Controller is thought of as a single entity. In the Custom Application Development scenario, we have divided the Controller into two components:

1. the View Controller, focused on the technical integration aspects of the Controller for the access channel (consisting of the Provider Router and Provider Adapter components)

2. the Business Controller, focused on the business process integration aspects of the Controller independent of access channel concern. (consisting of the Business Mediator, it's Business Mediator Functions, the Business Contract, and the Contract Validator)

Provider Router

Responsibility:

The Provider Router has two primary roles in R4SC: coordinate translation of incoming data through the appropriate Provider Adapter and direct traffic to the correct Business Mediator Function.

The Provider Router's traffic director responsibilities include: 1) knowing which Business Mediator's Function need to be invoked and in which order, and 2) respond accordingly as the SOA service defines. To accomplish the first responsibility, the Provider Router must do two tasks in order to trigger a Business Mediator's Function: 1) populate an instance of the Business Contract that is required by the Business Mediator's Function, and 2) invoke the Business Mediator's Function passing the contract as a parameter. For the second responsibility, the Provider Router must know how to trigger the appropriate SOA service response for all potential success and failure scenarios possible from the choreographed components. In order to enable the Application Client to deliver the display that is triggered, the Provider Router must also deliver to the Service Integrator the response information gathered while executing the service in the format specified by the SOA service definition.

Example: ATM Funds Transfer

In this example, we will assume that the Service has been routed by the ESB to the proper Service implementation, in this case the Provider Adapter. The caller has provided the source account id, the target account id, and the amount to transfer, according the the SOA Service definition. We will also assume that all transfers occur on the day requested.

The Provider Router must take the input in the format provided by the Service invocation. The Provider Router will call the Provider Adapter passing the input information, which will translate the input and instantiate an instance of the Transfer Business Contract (implementation) with the source and target account id, and the transfer amount provided. The Provider Router will then call the transfer Business Mediator Function on the Financial Transactions Business Mediator passing the populated instance of the Transfer Business Contract returned to it by the Provider Adapter.

Upon receiving the response from the transfer Business Mediator Function, the Provider Router will again trigger the Provider Adapter passing the result set to be translated back into the format required by the SOA Service definition. The Provider Adapter will return the transformed result set to the Provider Router, who will respond according to the SOA Service definition and terminate.

Architectural Decisions:

- A Provider Router function is limited to only routing messages to a single Process Mediator Function to ensure process logic does not get distributed across a variety of design elements.

Provider Adapter

Responsibilities:

Business Mediators are written with the intention of being used by a wide variety of callers, which will vary in their communication protocol and data format. So, we need a component that knows the format of the information that is coming in from the service consumer and has the knowledge to translate that data to the format of the Business Mediator layer. That component is the Provider Adapter. The format that must be translated to is the Business Contract of the Business Mediator Function that will be invoked by the Provider Router to fulfill the service invocation.

Business Contract

Responsibilty:
To understand the responsibility of the Business Contract, you must first understand who it is written for. The primary user of the Buisness Contract is the Business Mediator Function. It is defined to capture the information requests needed by the Business Mediator Function to fulfill its function. Business Contracts represents a specification independent of concrete implementation, consistent with the Interface design pattern, and when defined represent the list of operation signatures that the Business Mediator Function will need to request. Defining the Business Contract as a specification is critical to afford the Provider Router the flexibility to implement according to its architectural needs. When implemented, the Business Contract represents an independent object that encapsulates the answers to these requests and is passed to the Business Mediator Function by the Provider Router.

Referencing a Business Contracts, a specification, by the Business Mediator Function, rather than a concrete implementation enables the Business Mediator Function to service any number of requestors without becoming coupled to them or having to change to support them. This allows the Business Mediator Function to serve SOA consumers in the same manner it serves Java application clients.

Examples of available implementation alternatives:

1. implement the Interface as a lightweight object for distributed applications
2. in a non-distributed scenario, have the Provider Router itself implement the Interface to reduce instantiation overhead

Example: ATM Funds Transfer
The Transfer Business Contract is defined for use by the transfer Business Mediator Function on the Financial Transactions Business Mediator. For the transfer to be executed, the transfer function needs access to the source account, target account, and amount to be transferred. Therefore, the Transfer Business Contract defines three operation signatures: "public String getSourceAccountID()", "public String getTargetAccountID()", and "public Double getAmount()".

Architectural Decisions:
- A Business Contract encapsulate the information needs of the Business Mediator Function and ia a specifications rather than concrete implementations
- Business Contracts are only allowed to contain getter methods and inquiry methods (e.g., "is…")
- Business Contract methods can return native types, simple object types (e.g., String), and Immutable Interfaces
- Business Contracts should be designed for use by a single Business Mediator Function.

Business Mediator (and it's Business Mediator Functions)

Responsibilty:
The Business Mediator is the "brains of the operation". A Business Mediator should encapsulate the business process logic and rules of the Functional Component processes. The Business Mediator is responsible for making Business Objects, that are not (coupled) related, work together to provide a complete logical unit of work.

To accomplish this, the Business Mediator must work with the Business Model; however, we do not want the Business Mediator to become tied directly to the concrete Business Model, so we have the Business Mediator Functions interact with the Mutable Interface of the Business Model objects.

Why is it important to ensure that the Business Mediator doesn't become tied to one Business Model?

Through the experience of delivering and supporting solutions through the ever changing needs of the business, we have discovered that the reuse of the Business Mediators is desirable across the organization, however the different parts of the organization frequently cannot agree on a consistent Business Model. In order to be flexible to the different organizations' Business Model needs and allow the Business Mediator to be reused across those models, it became critical to ensure that the Business Mediators didn't interact with a specific implementation of a Business Model. So, the Mutable Interface was introduced. The Mutable Interface is a specification of the Business Model, which means it defines the method signatures that are required, but allows the implementation to be flexible (the most common way to accomplish this is with the Interface design pattern). By integrating the Business Mediator to a Mutable Interface, rather than a concrete Business Model, we can now move Business Mediators to work on top of other concrete Business Models, as long as the model realizes the Mutable Interface specifications.

Each operation (Business Mediator Function) on a Business Mediator must take a Business Contract as a parameter. Provider Routers will only be able to access Business Mediator Functions by providing an implementation of the appropriate Business Contract. The Business Mediator Function will query the provided Business Contract for information it needs to complete the requested service, and will not forward the contract to objects out of the Business Controller Layer. The Business Mediator Function integrates Mutable Interfaces and other Business Mediator Functions to deliver its' business process. The Business Mediator Function is responsible for knowing the functions that need to be called and the order in which they must be called. If a Mutable Interface function needs to be called and parameters passed, the Business Mediator Function is responsible for either passing information it has gathered from services it has triggered or requesting information from its' Business Contract. If another Business Mediator Function is to be called, the calling Business Mediator Function is responsible for instantiating the appropriate Business Contract implementation and populating the contract with the information it has from work it has performed or from its' Business Contract.

The Business Mediator Function is responsible for handling all exceptions that are returned from Mutable Interfaces, The Contract Validator, and other Business Mediator Functions it leverages to deliver its' business process. In the case that a recovery alternative does not exist for the exceptions, the Business Mediator Function is responsible for delivering business exceptions to the Provider Router that triggered it.

In order to eliminate data compatibility exceptions that may arise as a result of inappropriate input data, the Business Mediator Function may choose to leverage a Contract Validator to confirm the state of the Business Contract implementation's data prior to using the data in it's processing.

The Business Mediator Function rules focus on the business logic that pertains to the process and the interoperation of the unrelated Business Objects. Any logic that is specific to a particular Business Object only should not be moved to the Business Mediator Function domain.

Rules question:

In the exchange Business Mediator Function example above, should the exchange have the responsibility for determining whether the exchanged amount can be moved?

The exchange Business Mediator Function should NOT manage the business rule to determine if the target account (Account A) has enough money to withdrawal. That is the responsibility of Account A. Without this knowledge, the Account object would be incomplete. Account would not have the basic information to complete its withdrawal request.

However, the exchange Business Mediator Function should have the rules to determine if both provided source and target accounts are compatible accounts for exchange. The difference in the case of this rule is that the rule focuses on the exchange process itself, and whether the source and the target, which don't have reference to each other, are appropriate for an exchange. Only the Business Mediator Function can handle this rule.

Finally, in situations where logical units of work are required to successfully complete a business process, the Business Mediator Function requested by the Provider Router should open, execute, and close the logical unit of work within the execution of that process. This will ensure the Business Mediator Functions can operate as independent, standalone processes, or be integrated into larger grained Business Mediator Functions.

Example: ATM Funds Transfer

To this point in the example, the Provider Router has called the transfer method on the Financial Transactions Business Mediator and passed an instance of the Transfer Business Contract to the transfer method as its only parameter.

The transfer method has the option to validate the contract. In this example, the transfer method needs to confirm that the source and target account ids are not null. So, the transfer method will call the validate method on the Transfer Contract Validator object and pass the reference to the instance of the Transfer Business Contract that the requestor provided. The transfer service will need to be prepared to handle two exceptions from the validation: SourceAccountIDInvalidException and TargetAccountIDInvalidException. If either of these exceptions are encountered, the transfer service will terminate unsuccessfully and raise the exception to the Provider Router for handling.

In the event that the Transfer Business Contract is validated to be in good status, the transfer service will call the getTargetAccountID(), getSourceAccountID(), and getAmount() methods on the Transfer Business Contract to gain access to the information it needs to execute the transfer. Once it has answers to these requests, the Financial Transactions Business Mediator's transfer Business Mediator Function will:

1. Trigger the retrieveAccount method on the AccountFactory* passing the "target account id".
2. Trigger the retrieveAccount method on the AccountFactory* passing the "source account id".
3. Create a logic unit of work.
4. Call the withdrawl operation on the source account's Mutable Interface.
5. Call the deposit operation on the target account's Mutable Interface.
6. Call the createTransferReceipt on the ReceiptFactory* passing the "target account id", "source account id", and "amount", which will return a Receipt Immutable Interface.
7. Return the Transfer Receipt Immutable Interface to the requestor and close the logical unit of work.

** This example assumes the architect has assessed that the delivery and support teams' skills are mature enough to manage the complexity of the Factory pattern. If this were not the case, the alternative would have been to directly access the methods defined on the Business Object itself. See*

Domain Factory *for more details on pro's and con's of using the Factory concept.*

The transfer Business Mediator Function will need to handle one business exception that the Account Mutable Inteface's withdrawal method may raise: InsufficientFundsException. In the event that the InsufficientFundsException is raised, the transfer Business Mediator Function will terminate unsuccessfully and raise the InsufficientFundsException to the Provider Router for handling.

In this example, the source and target Accounts did not have a knowledge of each other. However, the transfer Business Mediator Function required the user to pass in its' Business Contract, which answered three questions: getSourceAccountID, getTargetAccountID, and getAmount. The Business Mediator Function encapsulates the logic to use this provided information to move the specified amount from the source to the target account.

The Business Mediator Function made decoupled objects work together to accomplish the business process requested.

Architectural Decisions:

- Business Mediator Functions must take as their only parameter a Business Contract
- Business Mediator Functions will return one of five values: void, an Immutable Interface, a collection of Immutable Interfaces, a native type/Simple object (e.g., boolean or String), a collection of native types/Simple objects
- Business Mediator's cannot pass reference to the Business Contract outside of the Business Controller Layer
- Business Mediator Functions talk to Business Objects through a Mutable Interface
- A Business Mediator Function can leverage other Business Mediator Functions to delivery the business process.
- Business Mediators are not intended to have a single Business Mediator Function

A Business Mediator Function encompasses a complex business process, can integrate other Business Mediator Functions and Business Object operations in delivering its' process, maintains its own internal state during the process, and manages a logical unit of work which once completed must terminate. For those familiar with J2EE, this would create a mental picture of a process object with a method that manages the logical unit of work of the business process. For those from an SOA perspective, does this sound like a choreographed composite service? Conceptually, one man's large grained composite process is another man's fine grained part of a larger composite process. If that process is exposed as a SOA service, it becomes known as a composite service. The R4SC is written independent of the implementation technologies that will be applied to deliver a final solution, therefore:

- a Business Mediator Function, in its simplest form, may be a POJO with in line code managing the process logic
- a Business Mediator Function may be a C# object integrating other Business Mediator Functions through their WSDL definition, via an Integration Mediator
- a Business Mediator Function may be a BPEL choreography integrating other SOA services to deliver the process

These are simply three examples of composite Business Mediator Functions with different levels of granularity. They are not the only examples of Business Mediator Functions, but they reinforce the importance of thinking of the R4SC components conceptually, and therefore, flexible to support various technologies and granularity.

Contract Validator

Responsibilty:
This component is used to validate the state or content of a Business Contract. The contract must adhere to this validation or an exception will be raised. The Contract Validator is invoked by the Business Mediator Function when needed.

The purpose of the validation is to confirm that the minimum input data has been provided. The Contract Validator should not become a single source for all rules validation for a business process. Business process oriented rules should remain in the Business Mediator Functions, while rules internal to the Business Object should remain within the Business Object.

Another temptation is to propagate data and enterprise application validation checks to the Contract Validator. This cannot be allowed, as this will tie the business logic of the application to the current back-end data sources and enterprise applications, making the solution less flexible to change. This typically becomes a temptation, because there is a desire to catch problems with the user provided information as soon as possible due to a perception that this will improve performance. It is critical to assess the non-functional requirements for performance to make sure that there is justifiable business reason to consider moving such validation to a higher layer in the application. In making the decision to move the validation up to a higher layer in the architecture, you are accepting a trade-off in maintainability and flexibility of the solution.

Our experience has shown that in most cases, the decision to move validation to a higher layer of the architecture was made before justifying through performance testing that the move was necessary in order to meet the performance requirements. Our experience also has shown where the performance tests are run to confirm/disprove the need to move such rules up in the architecture, in most cases the move is not required in order to meet performance requirements. What should this tell you? Most application teams that decide to move the rules up in the architecture make the trade-off (which results in less flexible, less maintainable solutions) when it wasn't necessary to meet performance requirements.

In Java-based solutions, as an example, the use of the Contract Validator has benefited projects that choose to use it by virtually eliminating null pointer exceptions in the solution.

Example: ATM Funds Transfer

As discussed in the Business Mediator section, the Financial Transactions Business Mediator's tranfer method has opted to validate its' Transfer Business Contract. To accommodate this, a Transfer Business Contract Valition object must be defined with the method validate(Transfer Business Contract). This method will call the Transfer Business Contract's getSourceAccountID() and confirm that it is not null. In the event that the value is null, the validate method will instantiate and raise the SourceAccountInvalidException. If the source account is valid, the validate method will next call the Transfer Business Contract's getTargetAccountID() and confirm that it is not null. In the event that the value is null, the validate method will instantiate and raise the TargetAccountInvalidException. If the target account is valid, the validation will complete and return void to the Financial Transactions Business Mediator's transfer method.

Architectural Decisions:
- **Prohibit the duplication of rules in the Contract Validator.**
- Contract Validation should be managed by a support object that the Business Services leverage, rather than a function of the Business Services themselves.

Business Model Layer:

The domain of this layer is focused on real world, fine grained business entities. The Business Object names should trace back to the terms used and understood by the business. From a technical perspective the design of each Business Object consists of three components:

- Business Object, a concrete implementation of the business domain object
- Mutable Interface, a specification of the business object public interface, independent of concrete implementation
- View Inteface, a restricted specification that only provides access to the inquiry operations of the public interface of the business object

Each component of the Business Model layer will first be discussed, and at the end of this section, we will walk through an example.

Business Object

Responsibilty:

Business Objects are the "real world" business entities that are used by Business Mediator Functions to complete complex business processes (e.g., in the Finance industry business objects would include an Account, a Customer, a Fund, etc). Business Objects are responsible for encapsulating the business rules that are pertinent to them internally.

Example of Business Object rules:

A Portfolio should know how to answer its' balance by asking each of its' investments for their individual balances and summing the answers.

They should not include business rules for a business process of which they are only a part.

Example of rules that are not appropriate for the Business Object:

An Account should not know how to execute a transfer since a transfer is not internal to that account only. It should know how to deposit money or withdrawal money, however. Functions that an Financial Transactions Business Mediator would leverage through the Account Mutable Interface to complete the transfer Business Mediator Function.

Each Business Object must have an Immutable Interface and Mutable Interface. The Business Object is responsible for implementing the Mutable Interface, which in turn is responsible for extending the Immutable Interface. Aggregate whole Business Objects should hold reference to the Mutable Interface of their aggregate part Business Object(s).

The Business Object is responsible for integrating with the Integration Adapter layer. Up to four steps are required to accomplish this:

1. The Business Object is responsible for instantiating and populating the Integration Contract required to trigger the required Integration Mediator Function.
2. The Business Object is responsible for triggering the required Integration Mediator Function passing the Integration Contract instantiated in step 1.
3. The Business Object is responsible for transferring the hydrating itself from the Immutable Data Object Interface returned by the Intergration Mediator Function.
4. In the event of a Exception, the Business Object is responsible for catching all Integration Exceptions and either handling them or throwing a business specific exception in the event that the Integration Exception cannot be handled.

Example: ATM Funds Transfer

Earlier we discussed the process of conducting a transfer, which is managed by the transfer Business Mediator Function. Now we will discuss what happens from a Business Object perspective when they are integrated by the Business Mediator Function. The key methods to discuss for this example are the Account Business Object's withdrawal, deposit, and hasSufficientFunds operations.

The Account Business Object must implement the Account Mutable Interface, which has an operation "public boolean hasSufficientFunds(Double anAmount)". The implementation of this operation will compare the Account's balance attribute value to anAmount being requested for withdrawal. If the balance value is greater than anAmount, the response from this method will be true, there are sufficient funds. Otherwise, in the event that the balance is not greater than the withdrawal amount (anAmount), the hasSufficientFunds method will instantiate and raise the InsufficientFundsException.

When we discussed the Business Mediator/Business Mediator Functions, it was the transfer Business Mediator Function's responsibility to request the sourceAccountID from the Transfer Busienss Contract, and calling the AccountFactory to findAccount for the sourceAccountID. The same process was completed to retrieve the target Account. Assuming we have retrieved the source and target account, the transfer Business Mediator Function must call the withdrawal operation on the sourceAccount's Account Mutable Interface. As we discussed above, the Account Mutable Interface is a specification of the Account, which means it does not have a concrete implementation. Recall that the Account Business Object implements the Account Mutable Interface, which means the instance of sourceAccount is an Account Business Object and an Account Mutable Interface. They are simply two views, specification and implementation, of the source Account with different levels of access to information about the instance. When the transfer Business Mediator Function calls the withdrawal operation on the Account Mutable Interface, the virtual machine in a Java example will automatically route the request to the concrete Account Business Object. The concrete implementation of Account handles the withdrawal operation by first confirming that it hasSufficientFunds, as we discussed above. If sufficient funds are present, the withdrawal operation will reduce the balance by the requeste amount, however, if sufficient funds are not present, the operation will complete by raising an exception. In the case of an exception, the transfer Business Mediator Function would halt it's logical unit of work and raise the exception to its' requestor. Assuming there is not an insufficient funds exception, the logical unit of work proceeds will continue with the request to deposit funds in the target Account, via the operation on the target Account Mutable Interface. The deposit operation simply adds to the balance of the target account, and returns control to the caller.

Finally, the transfer Business Mediator Function will request the creation of a Receipt to document the logical unit of work. To do this the transfer Business Mediator Function will call the createReceipt operation on the Receipt Business Object Factory, pass the target and source accountID's and transferAmount. The Receipt Business Object Factory will forward the request (and the source and target account Ids and transfer amount) to the create operation on the Transfer Receipt Business Object. To complete the create request, the Transfer Receipt Business Object will have to make a call to the createTransferReceipt Integration Mediator Function on the Financial Transactions Integration Mediator, which requires an instance of the Create Transfer Receipt Integration Contract. So, first the Transfer Receipt Business Object will request an instance of the Create Transfer Receipt Integration Contract be created by calling instantiate on the Create Transfer Receipt Integration Contract Implementation, passing the source and target account Ids and the transfer amount as input (an object Create Transfer Receipt Integration Contract Implementation will have to be developed that implements the Create Transfer Receipt Interface specification). With the newly created Create Transfer Receipt Integration Contract, the Transfer Receipt Business Object will then call the createTransferReceipt Integration Mediator Function. One of two results will be returned: a Transfer Receipt Immutable Data Object Interface or an exception. In the case of an exception, the process will stop and the Transfer Receipt Business Object will have to throw a business exception indicating the system failure. In the event that the create request is successful, the Transfer Receipt Business Object will call the instantiate operation on itself and pass in the Transfer Receipt Immutable Data Object Interface as a parameter. The instantiate operation will transfer the data from the Transfer Receipt Immutable Data Object Interface to its attributes (e.g., call getSourceAccountId on the Transfer Receipt Immutable Data Object Interface and set the data in it's sourceAccountId attribute). Finally, the newly created instance of the Transfer Receipt Business Object will be returned as a Trasfer Receipt Immutable interface by the Receipt Business Object Factory to the transfer Business Mediator Function.

As always, the Account Business Object must implement the Account Mutable Interface and the Receipt Business Object implements the Receipt Mutable Interface.

Architectural Decisions: ▪ Aggregate Business Objects hold reference to Mutable Interface of their aggregate parts ▪ BusinessObjects must know how to instantiate, retrieve, populate, and persist themselves ▪ Each BusinessObject must implement it's corresponding Mutable Interface ▪ Business Objects should have attributes on themselves to hold their data and maintain their state internally. ▪ Instance creation and management of the Business Object must be managed through a factory method.

Mutable Interface

Responsibility:
The Mutable Interface is a specification for a Business Object independent of concrete implementation, consistent with the Interface design pattern, and when defined represent the list of operation signatures that the Business Object will provide. The Mutable Interface has two primary responsibilities:

1. Provide access to inquiry and modify the state of a Business Object
2. Decouple the layers of the architecture from the Business Object's concrete implementation

The only users of the Mutable Interface are: Business Mediator/Business Mediator Functions and Business Objects.

From a UML relationship perspective, the Mutable Interface extends the Immutable Interface services with a list of method signatures that the user can call to change the public state of a Business Object without becoming coupled to the Business Object. The Mutable Interface evolved and was added to the R4SC in order to decouple the Business Mediator from the Business Model that it was originally built upon, thus allowing the Business Mediator to become a componentized asset that can be reused on top of other Business Models.

Point of Interest:

In order to integrate a Business Mediator on a different Business Model, the new Business Model must provide implementations for the Immutable Interfaces and Mutable Interfaces that the Business Mediator leverages.

The primary user of the Mutable Interface is the Business Mediator/Business Mediator Function. The Mutable Interface defines a view on the Business Model (Model) that the Business Mediator/Business Mediator Function (Controller) uses to modify the state of the Business Model. Other users of the Mutable Interface include: Business Objects. The Business Objects use the Mutable Interfaces to decouple themselves from the specific implementations of other Business Objects in the model. The aggregate whole Business Object will hold reference to and modify the state of its aggregate part Business Objects through the Mutable Interface, rather than directly through the Business Object.

It is… a specification defining operations independent of concrete implementation; provides extended access to modify the state of a business object

It is NOT… a specific concrete implementation of the operations defined; does not provide access to private or protected operations of the business object

Architectural Decisions:

- Mutable Interface may provide access to public setter methods, service methods that change state, and getter methods that the designer did not want to expose to the user interface but must expose to the Business Service and other Business Objects.
- Mutable Interface methods can return native types, simple object types (e.g., String), Mutable Interfaces, Immutable Interfaces, or void.
- From a UML perspective, the Mutable Interface extends the Immutable Interface of the Business Object.

Immutable Interface

Responsibilty:

The Immutable Interface is a specification which limits the user to operations to inquire about the state of a Business Object, but not change the state. As a specification, the Immutable Interface is independent of concrete implementation, consistent with the Interface design pattern, and when defined represent the list of operation signatures to inquire the state of the Business Object. When compared to the Mutable Interface, the Immutable Interface reduces the visibility to inquiry only operations, which allows this component to be returned to Views (as defined in Model View Controller terminology) and ensure compliance with the MVC framework.

The Immutable Interface has two primary responsibilities:

1. provide access to inquire (e.g., getter access) but not change (e.g., no setter access) the public internal state of a Business Model object
2. decouple the layers of the architecture from a Business Model object's specifics

The Immutable Interface represents the list of method signatures that the user can request in order to receive information about the public state of a Business Object without becoming coupled to the domain. Immutable Interfaces are implementations of the Interface design pattern and when defined represent the method signatures for the requests without their concrete implementations.

The primary user of the Immutable Interface is the Provider Router component. The Immutable Interface defines a view on the Domain Model (Model) that can be returned by the Business Mediator Functions(Controller) to the Provider Router (View) to preserve the integrity of the Model-View-Controller (MVC) framework.

Other users of the Immutable Interface include: Business Contracts, Business Mediator/Business Mediator Functionss, Business Objects, and Integration Contracts. All of these objects should make inquiry requests to the Immutable Interface of a Business Object, rather than talk directly to the Business Object.

It is… a specification defining operations independent of concrete implementation; provides extended access to modify the state of a business object

It is NOT… a specific concrete implementation of the operations defined; does not provide access to private or protected operations of the business object

<u>Architectural Decisions:</u> ▪ Immutable Interface will adhere to the guidelines of the Interface pattern and be implemented by it's Business Object ▪ Immutable Interfaces are only allowed to contain getter methods ▪ Immutable Interface getter methods can return native types, simple object types (e.g., String), or Immutable Interfaces ▪ In the case that a Business Service method is provided to initially retrieve a part of a Business Object, the Business Object's Immutable Interface should not expose the getter method for that part. ▪ Immutable Interfaces getter methods should not have any parameters.

Example: ATM Funds Transfer

As we discussed, the Mutable Interface represents the specification (definition without implementation) of the public interface for the Business Object. In addition, we discussed how the Immutable Interface provides a limited view to the public interface of the Business Object, limiting visibility to the public methods to inquire but not change the state of the Business Object. From a UML perspective, the most efficient way of capturing and reusing this information is to have the Mutable Interface extend the Immutable Interface.

So, now we will discuss the steps we would have taken when defining the Mutable and Immutable Interfaces for the transfer example. For the transfer example, we have discussed two Business Objects which must each have a Immutable and Mutable Interface: Account and Transfer Receipt. For the sake of this example, we will limit the discussion to the methods that these Facades will need to expose during the transfer process.

From an Account perspective, methods that would be appropriate for the transfer would be getBalance and getAccountId. These methods will be needed when the transfer Business Mediator Function requests a Receipt be created for the transfer.

The Transfer Receipt Immutable Interface will be the value returned by the transfer Business Mediator Function., therefore, it will need to provide methods to access details of the transaction. To deliver this information, the Transfer Receipt Immutable Interface will define the following operation: getSourceAccountId, getTargetAccountId, getAmount, getDate, and getConfirmationNumber.

Now let's shift our focus to the Mutable Interfaces. Three operations of the Account are involved in completing a transfer: withdrawal, deposit, and hasSufficientFunds. Both the withdrawal and deposit operations will need to be called by the transfer Business Mediator Function, and therefore, must be defined on the Account Mutable Interface. The hasSufficientFunds operation, however, is leveraged by the withdrawal operation to validate that there are adequate funds available to withdrawal. The hasSufficientFunds operation is needed by methods within the Account, but not appropriate for access by other objects. So, the hasSufficientFunds method will be defined on the Account Business Object, but not on the Account Mutable Interface, because it is not a part of the public interface. Remember when defining the Account Mutable Interface to establish the extends relationship from the Mutable Interface to the Immutable Interface, so that the Account Mutable Interface inherits all of the inquiry methods defined on the Immutable Interface.

Finally, the transfer Business Mediator Function needs to be able to populate the Transfer Receipt Business Object. To do this, the transfer Business Mediator Function will call the create operation on the Receipt Business Object Factory, and will pass as parameters the source Account ID, target Account ID, and transfer amount. The Receipt Business Object Factory will know to call the Transfer Receipt Business Object's constructor method and pass the data provided by the transfer Business Mediator Function, which will set the data on the object's appropriate attributes. For the sake of this example, there is really no need for the setter methods of the Transfer Receipt Mutable Interface to be called. Should the Transfer Receipt be able to be modified at some future time? Absolutely not, so, in this case we will not define any setter functions on the Transfer Receipt Mutable Interface.

Integration Layer

<u>Architectural Decisions:</u>

- Only the Integration Layer of the application can interface with the enterprise data sources.

Integration Contract

<u>Responsibilty:</u>
Integration Contracts represents a specification independent of concrete implementation, consistent with the Interface design pattern, and when defined represent the list of operation signatures that the Integration Mediator Function will need to request. Defining the Integration Contract as a specification is critical to afford the caller the flexibility to implement according to its architectural needs. When implemented, the Integration Contract represents an independent object that encapsulates the answers to these requests and is passed to the Integration Mediator Function by the caller.

Defining the Integration Mediator Function to reference an Integration Contracts, a specification rather than a concrete implementation, enables the Integration Mediator Function to service any number of requestors without becoming coupled to them or having to change to support them. It also provides the requestors flexibility in the way they architect the concrete implementations of the Integration Contract to support their unique architectural needs. This allows the Integration Mediator Function to serve SOA consumers in the same manner it serves Java application clients.

Example: ATM Funds Transfer

In our example, we have three primary business operations: withdrawal, deposit, and createReceipt. In simplest scenarios, this would equate to three Integartion Mediator Functions that would each need their own Integration Contract. For this example, we will keep it simple, and create a Withdrawal Integration Contract, Deposit Integration Contract, and Transaction Receipt Integration Contract.

The Withdrawal Integration Contract is defined for use by the withdrawal Integration Mediator Function defined on the Financial Transactions Integration Mediator. We'll assume this mediator function integrates to a DB2 table that requires the primary key for the source Account, the source account id, and the amount withdrawn. So, the Withdrawal Integration Contract will have two functions: getAccountId and getAmount.

The Deposit Integration Contract will look identical to the Withdrawal Integration Contract, as it will also require a getAccountId and getAmount operation. We often are asked, why then would we create two separate contracts with identical methods. The answer is maintainability and total cost of ownership of the system. (see, Architectural Decision IA4.0 - Integration Contracts should be designed for use by a single Integration Mediator service method)

The Transaction Receipt Integration Contract is defined for use by the createReceipt Integration Mediator Function. For the transfer to be documented, the createReceipt function needs access to the source account id, target account id, and amount to be transferred. Therefore, the Transaction Receipt Integration Contract defines three operations: getSourceAccountId, getTargetAccountId, and getAmount.

Architectural Decisions:

- Implementation of the Integration Contract can hold references to Immutable Interfaces, but cannot provide an accessor operation that returns the Immutable Interface
- Integration Contracts are only allowed to contain getter methods and inquiry methods (e.g., "is…")
- Integration Contracts should be designed for use by a single Integration Mediator Function.
- Integration Contracts encapsulate the information needs of the Integration Mediator Function and is a specifications rather than concrete implementations.
- Integration Contract methods can return native types and simple object types (e.g., String) only

Integration Mediator (and its' Integration Mediator Functions)

Responsibilty:

Integration Mediators encapsulate the business domain from the knowledge of the enterprise integration process being used for data storage and retrieval. The Integration Mediator is responsible for making Data Objects that are not related work together to provide a complete Integration Mediator. By encapsulating this logic from the domain, the integration technologies and the physical transactions can change with reduced impact to the requestor. Additionally, other domains can leverage the Integration Mediator Functions to gain access to the SOA services, operational systems and data store(s) it encapsulates.

The Integration Mediator should be written in such a way to provide the data needs for a business process, so that the requestor doesn't know the number of data sources and enterprise applications that are being touched to manage the data for that business process.

Therefore, a primary responsibility of the Integration Mediator Function is to manage the logical unit of work that spans the physical transactions of the disparate back-end systems and provides roll-back services in the event that one of those physical transactions fail. The logical unit(s) of work and all physical transactions must be started, executed, exceptions handled including rollback, and the unit of work closed within the process of the Integration Mediator Function.

Each Integration Mediator Function must take an Integration Contract as its' only parameter. Requestors will only be able to access Integration Mediator Functions by providing the appropriate Integration Contract. The Integration Mediator Function will query the provided Integration Contract for information it needs to complete the requested operation, and will not forward the contract to objects out of the Integration Mediator layer. The Integration Mediator Function may use Data Objects and other Integration Mediator Functions to deliver its' operation. The Integration Mediator Function is responsible for knowing the functions that need to be called and the order in which they must be called. If a Data Object's function needs to be called and parameters passed, the Integration Mediator Function is responsible for either passing information it has gathered from services it has triggered or requesting information from the instance of the Integration Contract that was passed to it. If another Integration Mediator Function is to be called, the calling Integration Mediator Function is responsible for instantiating the appropriate Integration Contract and populating the contract with the information it has from processes it has performed or from the Integration Contract that was passed to it. A Integration Mediator Function must return one of the following to its' users: void, a Integration Exception, an Immutable Data Object Interface, or a collection of the same type of Immutable Data Object Interfaces.

The Integration Mediator Function is responsible for handling all exceptions that are returned from Data Objects and other Integration Mediator Functions it leverages to deliver its' process. In the case that a recovery alternative does not exist for the caught exceptions, the Integration Mediator Function is responsible for delivering the technology and product (e.g., data source, middleware, and enterprise application) independent exceptions to the caller that triggered it.

Example: ATM Funds Transfer

To this point in the example, the Transfer Receipt Business Object has called the createTransferReceipt Integration Mediator Function on the Finacial Transactions Integration Mediator and passed an instance of the Create Transfer Receipt Integration Contract as its only parameter.

Now the createTransferReceipt Integation Mediator Function will call the getTargetAccountId(), getSourceAccountId(), and getAmount() methods on the Create Transfer Receipt Integration Contract to gain access to the information it needs to execute the request. Once it has answers to these requests, the createTransfer Receipt Integration Mediator Function will call the execute class method (passing the source an target account ids and transfer amount) on the Transfer Receipt Data Object.

Let's assume a simple scenario for this example, one where the information ultimately is stored in a relational database. With that assumption, the Transfer Data Object's execute method call an Integration Adapter which knows how to manage the communication with the target DB2 database. The Integration Adapter will as the Transfer Receipt Data Object to transform its attributes into the required format to make the SQL call, open the communication channel and execute the request on the DB2 database. The successful execution of this request will result in a respond stream with a String, the transfer confirmation number. The DB2 Integartion Adapter will close the connection to the database and returning the response stream to the Transfer Receipt Data Object. The Transfer Receipt Data Object will store the response data (in this case the transasction confirmation number) in its' attribute "transactionId". The Transfer Receipt Data Object will be returned to the createTransferReceipt Integration Mediator Function, which will in turn return the Transfer Receipt Immutable Data Object Interface to the Transfer Receipt Business Object that called it.

Architectural Decisions:

- All logical units of work to manage the integration of disparate back-end technology must be started, executed, and closed within the execution of the Integration Mediator process
- Integration Mediator service methods must take as their only parameter a Integration Contract
- Integration Mediator service methods can return one of five values: void, an Immutable Data Object Interface, a collection of Immutable Data Object Interface, a native type, a collection of native types
- Integration Mediator can pass reference to the Integration Contract to objects within the Integration Layer.
- An Integration Mediator Function can leverage other Integration Mediator Functions to delivery its operation.
- Integration Mediators are not intended to have a Integration Mediator Function

Immutable Data Object Interface

Responsibilty:

The Immutable Data Object Interface is a immutable specification for a Data Object independent of concrete implementation, consistent with the Interface design pattern, and when defined represent the list of inquiry operation that the Data Object will provide. The Immutable Data Object Interface has two primary responsibilities:

1. provide access to inquire (e.g., getter access) but not change (e.g., no setter access) the public internal state of a Data Object domain
2. decouple the layers of the architecture from a Data Model object's specifics

The Immutable Data Object Interface represents the list of method signatures that a requestor can access in order to receive information about the public state of a Data Object without becoming coupled to the Integration Data model. The users of the Immutable Data Object Interface include: Integration Mediator Functions and Business Objects.

The primary user of the Immutable Data Object Interface is the Business Object. The Immutable Data Object Interface defines a view on the Integration domain model that can be returned by the Integration Mediator to the requestor. This is a different twist on the application of the Model-View-Controller (MVC) framework, in which the requesting system represents the View, the Integration Mediator is the Controller, and the Data Objects are the Model.

It is… a specification defining operations independent of concrete implementation; provides limited access to inquire the state of a Data Object

It is NOT… a specific concrete implementation of the operations defined; does not provide access to private or protected operations of a Data Object

Example: ATM Funds Transfer

In the case of the withdrawal and deposit Integration Mediator Functions, they execute upon the Account Data Object, but there is nothing of significance for this example related to the Account Data Object. The most interesting Data Object for this example is the Transfer Receipt Data Object, which will require an Immutable Data Object Interface to be returned by the create transfer receipt Integration Mediator Function. The Transfer Receipt Data Object consists of the source Account Id, target Account Id, amount transferred, date of transfer, and a unique transaction number. To allow the caller to gain access to this information, but not change the state of the Data Object, we must create an Transfer Receipt Immutable Data Object Interface. The immutable interface will provide the getSoureceAccountId, getTargetAccountId, getTransferAmount, getTransactionDate, and getTransactionIdentifier operations. The Transfer Receipt Data Object will need to implement this immutable specification according to the architectural decisions for that component.

Architectural Decisions:

- Immutable Data Object Interface will adhere to the guidelines of the Interface pattern and be implemented by it's Data Object
- Immutable Data Object Interfaces are only allowed to contain getter methods and inquiry methods ("is…").
- Immutable Data Object Interface getter methods can return native types, simple object types (e.g., String), or other Immutable Data Object Interfaces
- Immutable Data Object Interfaces getter methods should not have any parameters.

Data Object

Responsibilty:

The domain of the Integration Mediator is integration to back-end data sources. Therefore, the model of the Integration Mediator is focused on the domain of data, whereas, the domain of the Business Objects is business logic.

Each Data Object must have and implement an Immutable Data Object Interface. Aggregate whole Data Objects should hold onto their aggregate part Data Object references as the parts Immutable Data Object Interface. Aggregate whole Data Objects should hold reference to the Immutable Data Object Interface of their aggregate part Data Object(s).

The Data Object is responsible for:

1. Mapping the data contained in its attributes to the format required by the integrated APIs.

2. Mapping the response data back to its' attributes and aggregate Data Objects.
3. In the case that multiple operational systems, middleware components, databases are required to retrieve or persist data for a Data Object, the Data Object is responsible for managing the logical unit of work for its data integration within its' operation execution.

Example: ATM Funds Transfer

The final part to this example is the Transfer Reciept Data Object. In the Integration Mediator section, we discussed the Integration Mediator calling the execute method on the Transfer Receipt Data Object. That method is defined as, execute(sourceAccountId, targetAccountId, amount, transferDate).

The Transfer Data Object is responsible for mapping this input into the format of the integrated componet (e.g., database, SOA Service, operational system API) that is being leveraged for the application. Once the data is in the format, the Transfer Receipt Data Object will call the appropriate Integration Adapter to open communication to the integrated technology and forward the request. The Transfer Receipt Data Object is then responsible for transforming the response into the return format of the execute method. In this example, this process is simple since the return is a String. In a more complex example, the Data Object would need to transform the response into the appropriate Data Object format and return an Immutable Data Object Interface.

Architectural Decisions:

- Aggregate Data Objects hold reference to Immutable Data Object Interface of parts (can cast reference to it's Data Object temporarily within the execution of it's methods, when needed.)
- Data Objects must know how to instantiate, retrieve, populate, and persist themselves
- Instance creation and management of the Data Object must be managed through a factory method.

Integration Adapter

Responsibilities:

The Integration Adapter is responsible for encapsulating the technology integration. The Integration Adapter must know how to manage the common functionality of integrating with a back-office technology. For example, there are common capabilities that are required in order to establish a connection and communicate with a relational database, which if centralized would greatly streamline the code base, as well as ensure consistency. From an SOA solution stack perspective, the Integration Adapter is responsible for managing the communication with the Operational Systems layer, Integration layer, and Data Architecture layer. The Integration Adapter is responsible for:

1. Open the communication channel and connect to the back-end applications.
2. Managing the physical transaction (Open, execute, handle exceptions including rollback, and close) within it's method execution.

Mapping the R4SC Custom Application Development Scenario to the SOA Solution Stack (S3)

As we discussed at the start of this document, the R4SC addresses the Service Component layer of the S3. The Provider Router is the entry point into the service component, as it implements the Service definition and encompasses the binding to the appropriate Business Mediator Function, which provides the micro flow of the service component. The components of the Custom Application Development Scenario, between the Business Mediator and the Integration Adapter, are contained within the black box of the Service Component. Working together, these components fulfill their prescribed responsibilities, described above, to ensure the delivery of the Service Component and maintain separation of responsibility, so that the Service Component remains adaptable to business change. The image below captures the alignment of the components to the S3.

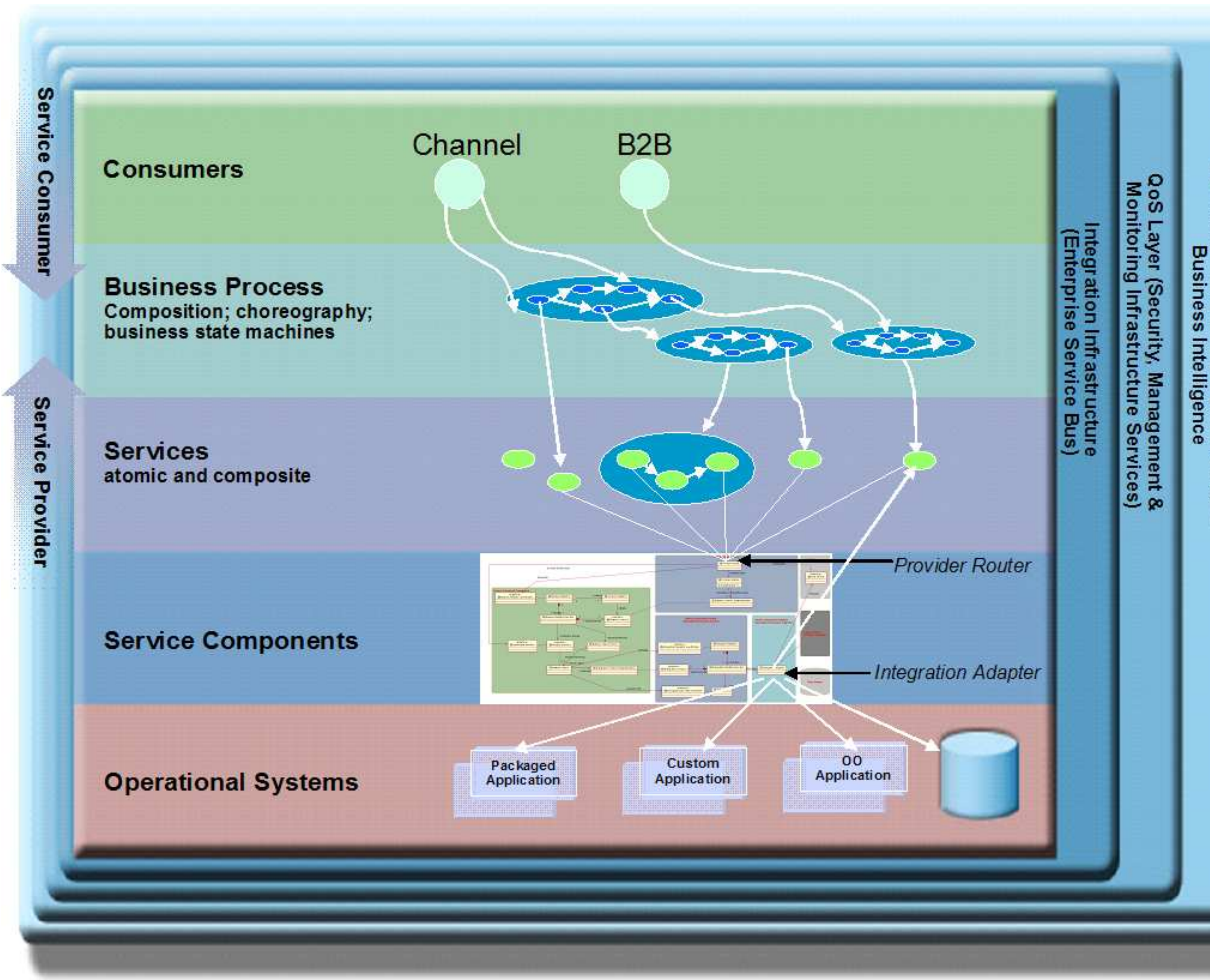


Figure 4 - R4SC Custom Application Development Scenario mapping to SOA Solution Stack (S3)

Integration Development Scenario

The Integration Development Scenario focuses on the application of the R4SC patterns where the existing operational systems functionality provides, out of the box, the functionality to address the SOA service needs of the organization. As a result, there is a no need to add additional business and process logic in order to fulfill the demands of the business, the operational systems' functionality simply needs to be made available via SOA architecture. As a result there is a two tiered service:

1. The Service definition and Component layer – implementing the SOA service definition, translating to and from the technology format of the underlying application, and routing the service request to the application elements
2. The Integration Mediator Layer – encapsulating the data entities and technology integration to access the operational systems, data architecture, and integration technologies of the SOA Stack.

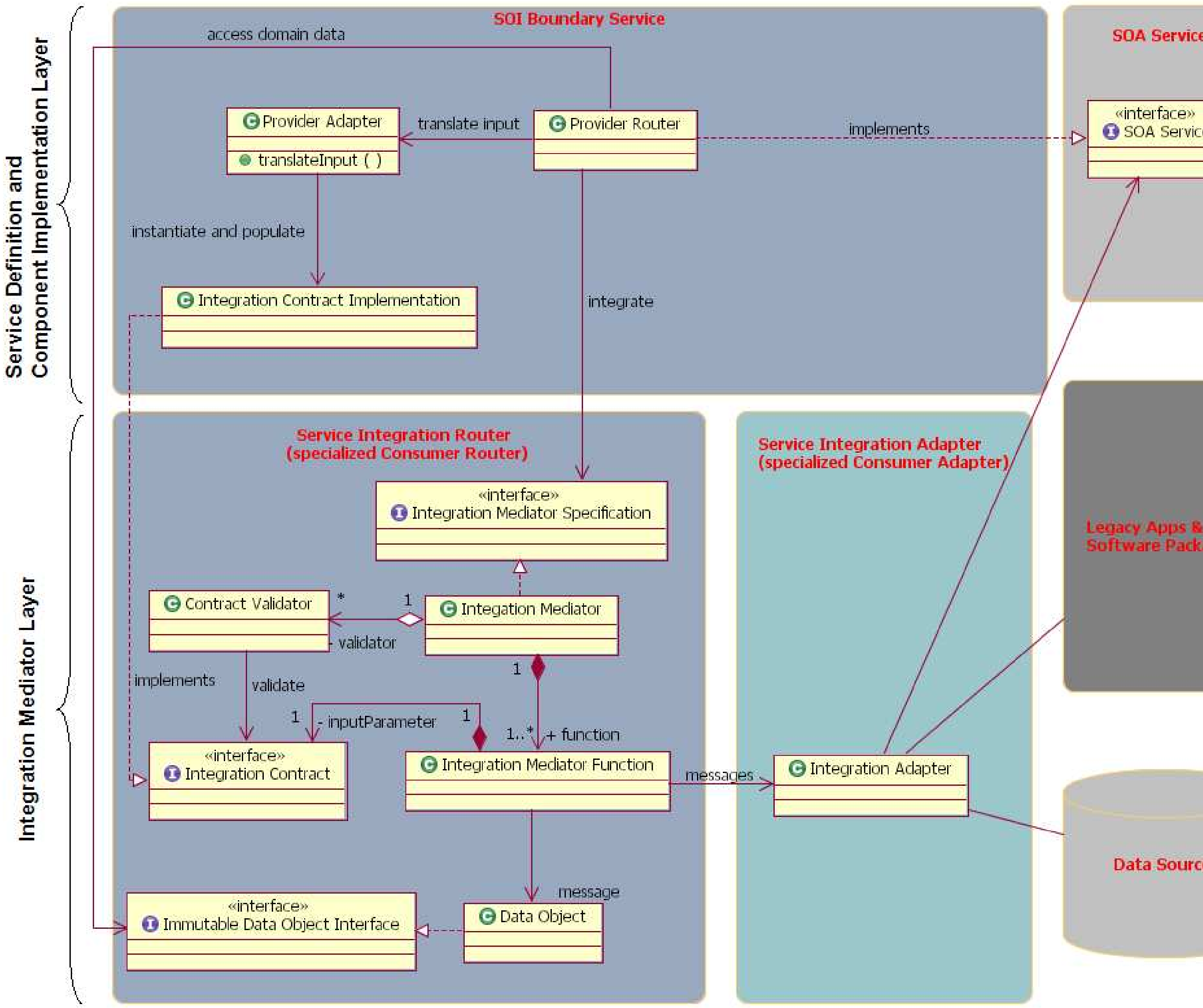


Figure 5 - R4SC, Integration Development Scenario

Service Definition and Component Implementation Layer:

Provider Router

Responsibility:

The Provider Router has two primary roles in R4SC: coordinate translation of incoming data through the appropriate Provider Adapter and directing traffic to the correct Integration Mediator Function.

The Provider Router's traffic director responsibilities include: 1) knowing which Integration Mediator's Function need to be invoked, and 2) respond accordingly to the consumer as the SOA service defines. To accomplish the first responsibility, the Provider Router must do two tasks in order to trigger an Integration Mediator's Function: 1) populate an instance of the Integration Contract that is required by the Integration Mediator's Function, and 2) invoke the Integration Mediator's Function passing the contract as a parameter. For the second responsibility, the Provider Router must know how to trigger the appropriate SOA service response for all potential success and failure scenarios possible, providing response information gathered in the format specified by the SOA service definition.

Example: ATM Funds Transfer

In this example, we will assume that the Service has been routed by the ESB to the proper Service implementation, in this case the Provider Router. The caller has provided the source account id, the target account id, and the amount to transfer, according the the SOA Service definition. We will also assume that all transfers occur on the day requested.

The Provider Router must take the input in the format provided according to the Service definition. The Provider Router will call the Provider Adapter passing the input information, which will translate the input and instantiate an instance of the Transfer Integration Contract (implementation) with the source and target account id, and the transfer amount provided. The Provider Router will then call the transfer Integration Mediator Function on the Financial Transactions Integration Mediator passing the populated instance of the Transfer Integration Contract returned to it by the Provider Adapter.

Upon receiving the response from the transfer Integration Mediator Function, the Provider Router will again trigger the Provider Adapter passing the result set to be translated back into the format required by the SOA Service definition. The Provider Adapter will return the transformed result set to the Provider Router, who will respond according to the SOA Service definition and terminate.

Provider Adapter

Responsibilities:

Integration Mediators are written with the intention of being used by a wide variety of callers, which will vary in their communication protocol and data format. So, we need a component that knows the format of the information that is coming in from the service consumer and has the knowledge to translate that data to the format of the Integration Mediator layer. That component is the Provider Adapter. The format that must be translated to is the Integration Contract required by the Integration Mediator Function that will be invoked by the Provider Router to fulfill the service invocation.

Integration Mediator Layer

Architectural Decisions:

- Only the Integration Layer of the application can interface with the enterprise data sources.

Integration Contract

Responsibilty:

Integration Contracts represents a specification independent of concrete implementation, consistent with the Interface design pattern, and when defined represent the list of operation signatures that the Integration Mediator Function will need to request to fulfill its operation. Defining the Integration Contract as a specification is critical to afford the caller the flexibility to implement according to its architectural needs. When implemented, the Integration Contract represents an independent object that encapsulates the answers to these requests and is passed to the Integration Mediator Function by the caller.

Defining the Integration Mediator Function to reference an Integration Contract, a specification rather than a concrete implementation, enables the Integration Mediator Function to service any number of requestors without becoming coupled to them or having to change to support them. It also provides the requestors flexibility in the way they architect the concrete implementations of the Integration Contract to support their unique architectural needs. This allows the Integration Mediator Function to serve SOA consumers in the same manner it serves Java application clients.

Example: ATM Funds Transfer

In our example, we have three primary business operations: withdrawal, deposit, and createReceipt. In simplest scenarios, this would equate to three Integartion Mediator Functions that would each need their own Integration Contract. For this example, we will keep it simple, and create a Withdrawal Integration Contract, Deposit Integration Contract, and Transaction Receipt Integration Contract.

The Withdrawal Integration Contract is defined for use by the withdrawal Integration Mediator Function defined on the Financial Transactions Integration Mediator. We'll assume this mediator function integrates to a DB2 table that requires the primary key for the source Account, the source account id, and the amount withdrawn. So, the Withdrawal Integration Contract will have two functions: getAccountId and getAmount.

The Deposit Integration Contract will look identical to the Withdrawal Integration Contract, as it will also require a getAccountId and getAmount operation. We often are asked, why then would we create two separate contracts with identical methods. The answer is maintainability and total cost of ownership of the system. (see, Architectural Decision IA4.0 - Integration Contracts should be designed for use by a single Integration Mediator service method)

The Transaction Receipt Integration Contract is defined for use by the createReceipt Integration Mediator Function. For the transfer to be documented, the createReceipt function needs access to the source account id, target account id, and amount to be transferred. Therefore, the Transaction Receipt Integration Contract defines three operations: getSourceAccountId, getTargetAccountId, and getAmount.

<u>Architectural Decisions:</u>

- Integration Contracts are only allowed to contain getter methods and inquiry methods (e.g., "is…")
- Integration Contracts should be designed for use by a single Integration Mediator Function.
- Integration Contracts encapsulate the information needs of the Integration Mediator Function and is a specifications rather than concrete implementations.
- Integration Contract methods can return native types and simple object types (e.g., String) only

Integration Mediator (and its' Integration Mediator Functions)

<u>Responsibilty:</u>

Integration Mediators encapsulate the caller from the knowledge of the enterprise integration process being used for data storage and retrieval. The Integration Mediator is responsible for making Data Objects that are not related work together to provide a complete Integration Mediator Function. By encapsulating this logic from the domain, the integration technologies and the physical transactions can change with reduced impact to the requestor. Additionally, other domains can leverage the Integration Mediator Functions to gain access to the SOA services, operational systems and data store(s) it encapsulates.

The Integration Mediator should be written in such a way to provide the data needs for a business process, so that the requestor doesn't know the number of data sources and enterprise applications that are being touched to manage the data for that business process.

Therefore, a primary responsibility of the Integration Mediator Function is to manage the logical unit of work that spans the physical transactions of the disparate back-end systems and provides roll-back services in the event that one of those physical transactions fails. The logical unit(s) of work and all physical transactions must be started, executed, exceptions handled including rollback, and the unit of work closed within the process of the Integration Mediator Function.

Each Integration Mediator Function must take an Integration Contract as its' only parameter. Requestors will only be able to access Integration Mediator Functions by providing an implementation of the appropriate Integration Contract. The Integration Mediator Function will query the provided Integration Contract for information it needs to complete the requested operation, and will not forward the contract to objects out of the Integration Mediator layer. The Integration Mediator Function may use Data Objects and other Integration Mediator Functions to deliver its' operation. The Integration Mediator Function is responsible for knowing the functions that need to be called and the order in which they must be called. If a Data Object's function needs to be called and parameters passed, the Integration Mediator Function is responsible for either passing information it has gathered from services it has triggered or requesting information from the instance of the Integration Contract that was passed to it. If another Integration Mediator Function is to be called, the calling Integration Mediator Function is responsible for instantiating the appropriate Integration Contract and populating the contract with the information it has from processes it has performed or from the Integration Contract that was passed to it. An Integration Mediator Function must return one of the following to its' users: void, a integration exception, an Immutable Data Object Interface, or a collection of the same type of Immutable Data Object Interfaces.

The Integration Mediator Function is responsible for handling all exceptions that are returned from Data Objects and other Integration Mediator Functions it leverages to deliver its' process. In the case that a recovery alternative does not exist for the caught exceptions, the Integration Mediator Function is responsible for delivering a technology and product independent exceptions to the caller that triggered it (e.g., no reference to data source, middleware, and enterprise application specific exceptions that would force the caller to be coupled to something technology specific).

Example: ATM Funds Transfer

To this point in the example, the Provider Router has called the createTransferReceipt Integration Mediator Function on the Finacial Transactions Integration Mediator and passed an instance of the Create Transfer Receipt Integration Contract as its only parameter.

Now the createTransferReceipt Integration Mediator Function will call the getTargetAccountId(), getSourceAccountId(), and getAmount() methods on the Create Transfer Receipt Integration Contract to gain access to the information it needs to execute the request. Once it has answers to these requests, the createTransfer Receipt Integration Mediator Function will call the execute class method on the Transfer Receipt Data Object, passing the source account id, target account id, and transfer amount.

Let's assume a simple scenario for this example, one where the information ultimately is stored in a relational database. With that assumption, the Transfer Data Object's execute method calls an Integration Adapter which knows how to manage the communication with the target DB2 database. The Integration Adapter will ask the Transfer Receipt Data Object to transform its attributes into the required format to make the SQL call. Then the DB2 Integration Adapter will open the communication channel and execute the request on the DB2 database. The successful execution of this request will result in a respond stream with a String, the transfer confirmation number. The DB2 Integartion Adapter will close the connection to the database and return the response stream to the Transfer Receipt Data Object. The Transfer Receipt Data Object will store the response data (in this case the transasction confirmation number) in its' attribute "transactionId". The Transfer Receipt Data Object will be returned to the createTransferReceipt Integration Mediator Function, which will in turn return the Transfer Receipt Immutable Data Object Interface to the Provider Router that called it.

<u>Architectural Decisions:</u>

- All logical units of work to manage the integration of disparate back-end technology must be started, executed, and closed within the execution of the Integration Mediator process
- Integration Mediator service methods must take as their only parameter a Integration Contract
- Integration Mediator service methods can return one of five values: void, an Immutable Data Object Interface, a collection of Immutable Data Object Interface, a native type, a collection of native types
- Integration Mediator can pass reference to the Integration Contract to objects within the Integration Layer.
- An Integration Mediator Function can leverage other Integration Mediator Functions to delivery its operation.
- Integration Mediators are not intended to have a single Integration Mediator Function

Contract Validator

<u>Responsibilty:</u>
This component is used to validate the state or content of an Integration Contract. The contract must adhere to this validation or an exception will be raised. The Contract Validator is invoked by an Integration Mediator Function when needed.

The purpose of the validation is to confirm that the minimum input data has been provided. The Contract Validator should not become a single source for all rules validation for an Integration Mediator Function. A common temptation is to propagate data and enterprise application validation checks to the Contract Validator. This cannot be allowed, as this will increase the coupling of the Integration Layer to the integrated data sources, operational systems, and integrated SOA services, making the solution less flexible to change. This typically becomes a temptation, because there is a desire to catch problems with the user provided information as soon as possible due to a perception that this will improve performance. It is critical to assess the non-functional requirements for performance to make sure that there is justifiable business reason to consider moving such validation to a higher layer in the architecture. In making the decision to move the validation up to a higher layer in the architecture, you are accepting a trade-off in maintainability and flexibility of the solution.

Our experience has shown that in most cases, the decision to move validation to a higher layer of the architecture was made before justifying through performance testing that the move was necessary in order to meet the performance requirements. Our experience also has shown when the performance tests are run to confirm/disprove the need to move such rules up in the architecture, in most cases the move is not required in order to meet performance requirements. What should this tell you? Most application teams that decide to move the rules up in the architecture make the trade-off (which results in less flexible, less maintainable solutions) when it wasn't necessary to meet performance requirements.

Example: ATM Funds Transfer

The Financial Transactions Integration Mediator's tranfer method has opted to validate its' Transfer Integration Contract. To accommodate this, a Transfer Integration Contract Valitor object must be defined with the method validate(TransferIntegrationContractAgreement). This method will call the Transfer Integration Contract's getSourceAccountID() and confirm that it is not null. In the event that the value is null, the validate method will instantiate and raise the SourceAccountInvalidException. If the source account is valid, the validate method will next call the Transfer Integration Contract's getTargetAccountID() and confirm that it is not null. In the event that the value is null, the validate method will instantiate and raise the TargetAccountInvalidException. If the target account is valid, the validation will complete and return void to the Financial Transactions Integration Mediator's transfer method.

<u>Architectural Decisions:</u>
- Prohibit the duplication of rules in the Contract Validator.
- Contract Validation should be managed by a support object that the Integration Mediator Functions' leverage, rather than a function of the Integration Mediators themselves.

Immutable Data Object Interface

<u>Responsibilty:</u>
The Immutable Data Object Interface is an immutable specification for a Data Object independent of concrete implementation, consistent with the Interface design pattern, and when defined represents the list of inquiry operation that the Data Object will provide. The Immutable Data Object Interface has two primary responsibilities:

1. provide access to inquire (e.g., getter access) but not change (e.g., no setter access) the public internal state of a Data Object domain
2. decouple the layers of the architecture from Integration Data model object specifics

The Immutable Data Object Interface represents the list of method signatures that a requestor can access in order to receive information about the public state of a Data Object without becoming coupled to the Integration Data model. The users of the Immutable Data Object Interface include: Integration Mediator Functions, Provider Router, and Provider Adapter.

The primary user of the Immutable Data Object Interface is the Provider Adapter. The Immutable Data Object Interface defines a view on the Integration domain model that can be returned by the Integration Mediator to the requestor.

It is… a specification defining operations independent of concrete implementation; provides limited access to inquire the state of a Data Object

It is NOT… a specific concrete implementation of the operations defined; does not provide access to private or protected operations of a Data Object

Example: ATM Funds Transfer

In the case of the withdrawal and deposit Integration Mediator Functions, they execute upon the Account Data Object, but there is nothing of significance for this example related to the Account Data Object. The most interesting Data Object for this example is the Transfer Receipt Data Object, which will require an Immutable Data Object Interface to be returned by the create transfer receipt Integration Mediator Function. The Transfer Receipt Data Object consists of the source Account Id, target Account Id, amount transferred, date of transfer, and a unique transaction number. To allow the caller to gain access to this information, but not change the state of the Data Object, we must create a Transfer Receipt Immutable Data Object Interface. The immutable interface will provide the getSoureceAccountId, getTargetAccountId, getTransferAmount, getTransactionDate, and getTransactionIdentifier operations. The Transfer Receipt Data Object will need to implement this immutable specification according to the architectural decisions for that component.

Architectural Decisions:

- Immutable Data Object Interface will adhere to the guidelines of the Interface pattern and be implemented by it's Data Object
- Immutable Data Object Interfaces are only allowed to contain getter methods and inquiry methods ("is…").
- Immutable Data Object Interface getter methods can return native types, simple object types (e.g., String), or other Immutable Data Object Interfaces
- Immutable Data Object Interfaces getter methods should not have any parameters.

Data Object

Responsibilty:

The domain of the Integration Mediator is integration to back-end data sources. Therefore, the model of the Integration Mediator is focused on the domain of data (e.g., no business logic, just data).

Each Data Object must have and implement an Immutable Data Object Interface. Aggregate whole Data Objects should hold reference to the Immutable Data Object Interface of their aggregate part Data Object(s).

The Data Object is responsible for:

1. Mapping the data contained in its attributes to the format required to call the integrated APIs.
2. Mapping the response data back to its' attributes and aggregate Data Objects.
3. In the case that multiple operational systems, middleware components, databases are required to retrieve or persist data for a Data Object, the Data Object is responsible for managing the logical unit of work for its data integration within its' operation execution.

Example: ATM Funds Transfer

The final part to this example is the Transfer Reciept Data Object. In the Integration Mediator section, we discussed the Integration Mediator calling the execute method on the Transfer Receipt Data Object. That method is defined as execute(sourceAccountId, targetAccountId, amount, transferDate).

The Transfer Data Object is responsible for mapping this input into the format of the integrated componet (e.g., database, SOA Service, operational system API) that is being leveraged for the application. Once the data is in the format, the Transfer Receipt Data Object will call the appropriate Integration Adapter to open communication to the integrated technology and forward the request. The Transfer Receipt Data Object is then responsible for transforming the response into the return format of the execute method. In this example, this process is simple since the return is a String. In a more complex example, the Data Object would need to transform the response into the appropriate Data Object format and return an Immutable Data Object Interface.

Architectural Decisions:

- Aggregate Data Objects hold reference to the Immutable Data Object Interface of its aggregate parts (can cast reference to it's Data Object temporarily within the execution of it's methods, when needed.)
- Data Objects must know how to instantiate, retrieve, populate, and persist themselves
- Instance creation and management of the Data Object must be managed through a factory method.

Integration Adapter

Responsibilities:
The Integration Adapter is responsible for encapsulating the technology integration. The Integration Adapter must know how to manage the common functionality of integrating with a back-office technology (e.g., data source, operational system, SOA Service, etc.). For example, there are common capabilities that are required in order to establish a connection and communicate with a relational database, which if centralized would greatly streamline the code base, as well as ensure consistency. From an SOA solution stack perspective, the Integration Adapter is responsible for managing the communication with the Operational Systems layer, Integration layer, and Data Architecture layer. The Integration Adapter is responsible for:

1. Opening the communication channel and connecting to the back-end applications.
2. Managing the physical transaction (Open, execute, handle exceptions including rollback, and close) within it's method execution.

Mapping the R4SC Integration Development Scenario to the SOA Solution Stack (S3)

As we discussed at the start of this document, the R4SC addresses the Service Component layer of the S3. The Provider Router is the entry point into the service component, as it implements the Service definition and encompasses the binding to the appropriate Integration Mediator Function, which provides the micro flow of the service component. The components of the Integration Development Scenario, between the Integration Mediator and the Integration Adapter, are contained within the black box of the Service Component. Working together, these components fulfill their prescribed responsibilities, described above, to ensure the delivery of the Service Component and maintain separation of responsibility, so that the Service Component remains adaptable to business change. The image below captures the alignment of the components to the S3:

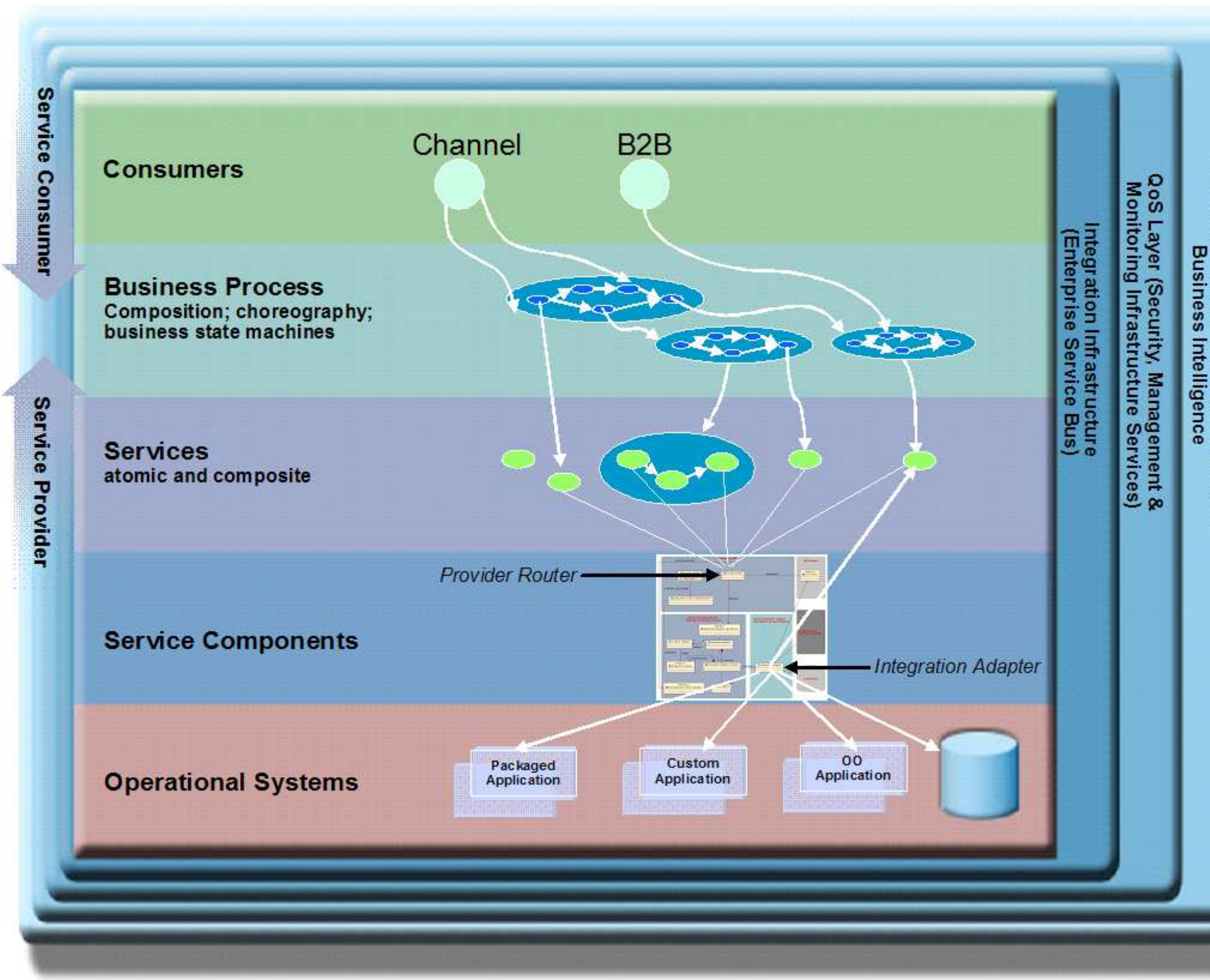


Figure 6 - R4SC Integration Development Scenario mapping to the SOA Solution Stack (S3)

Appendix:

Architectural Decisions: R4SC Service Component Pattern

Process Contract

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Process Contracts encapsulate the information needs of the Mediator Function and are abstract implementations of the Interface pattern.	BC1.0	Business Controller
Problem Statement	How can requests for information that the Mediator Function requires be answered in order to perform the requested service, while providing maximum flexibility in the implementation of those requests?		
Assumptions	▪ Mediator Functions must be independent of knowledge of those requesting its service. ▪ Implementation flexibility is critical to allow users to take advantage of various application distribution architectures and technologies.		
Motivation	Design a Process Mediator that can be leveraged across requestors/usage scenarios without a need to change the process logic implementation; thus, ▪ Ensuring consistency of response across access channels through reuse, ▪ Reducing maintenance by eliminating redundant silos of business logic per access channel.		
Alternatives	1. Create a concrete class that encapsulates the requests for information and place an instance of this class as the parameter for the Mediator Function. **2. Leverage the interface pattern to create an abstract definition of the request for information that the Mediator Function expects answered. Place an instance of this interface as the parameter for the Mediator Function.**		

Decision	Option 2 was selected, because it allows for implementation flexibility, thus better addressing the second assumption.

Additionally, Option 2 allows us to define a specification which limits the Process Mediator Function to access methods that only inquire the state of the Process Contract. With Option 2, the implementation specifics of the class that implements the Process Contract are not visible to the Process Mediator. Whereas, Option 1 doesn't take advantage of an interface, instead defining a class implementation of the Process Contract and having an instance of this class be the input parameter for the Process Mediator Function. Unfortunately, with Option 1, the Process Mediator would have access to methods that would allow it to change the state of the Process Contract, which it should not do. Option 2 prevents this from happening. Why is this important? It prevents the Process Mediator from changing the state of the contract, in other words, it prevents the input data from possibly being changed by the user of the data.

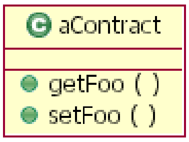
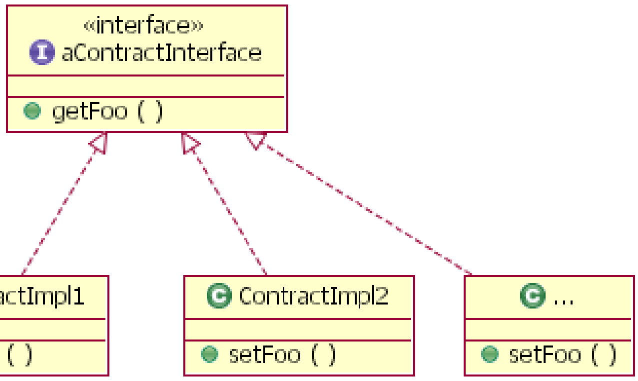


Figure 4 - Option 1



Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Process Contracts are only allowed to contain getter methods and inquiry methods (e.g., "is…")	BC2.0	Business Controller
Problem Statement	Should the contract be used as a unidirectional transport of information to the Mediator Function or should the contract be allowed to be a bidirectional transport of information between the Mediator Function and its requestor?		
Assumptions	• The standard protocol for returning information in OO languages is to use the return value to return requested information from a service to its requestor.		
Motivation	A goal of the Process Contract is to provide a simple, user friendly API for users of the Mediator Function. The solution should require as little documentation as possible for a user to understand the appropriate way to integrate it. The degree		

	to which optional information that depends on domain state/usage can be reduced/eliminated adds to the simplicity of the solution.
Alternatives	1. Allow the Process Contract to contain setter methods, thus allowing the Mediator Function to add information gathered during its execution. **2. Allow the Process Contract to provide only getter methods, thus reducing the contract to a unidirection vessel for passing information to the Mediator Function for the operation's consumption.**
Decision	Option 1 was not selected, because it complicates the usability of the contract. This approach results in contracts with getter and setter methods defined for two different users: the Mediator Function and the requestor. What information must the requestor provide to the Mediator Function for it to function successfully? Which of the getters is provided for the requestor to receive information? When can the requestor expect to be able to call the getter methods whose data is populated by the Mediator Function and have valid information? These are all questions that are raised by the integrator of a Mediator Function when the Process Contract was allowed to have setters. This resulted in additional support demand on the Process Mediator owner, and delays experienced by the integrator. Both resulting in project delays. By eliminating the setters, we eliminate the questions, which eliminates the delays. **Option 2 was selected**, because it provides the most user friendly alternative. The contract has only one user: the Mediator Function. All the getter methods on the Process Contract are expected by the Mediator Function, and therefore, must be provided by the requestor (no question as to who provides information and when it needs to be available). Forces the Mediator Function to use the return value to provide information back to the requestor, which is a better practice than placing return information on the contract.

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Process Contract methods can return native types, simple object types (e.g., String), and Immutable Interfaces	BC3.0	Business Controller
Problem Statement	How can the potential return types of the contract that the requestor (view… a non-visual integrator is a view in the Model-View-Controller sense) will have access to be minimized so that the model-view-controller (MVC) principles are preserved?		
Assumptions	• Users of the Mediator Function will not have access to the Model or components that can directly affect changes to the model without going through the controller.		
Motivation	Adherence to model-view-controller (MVC) will reduce impact of change to the implementation throughout the life of the application, thus reducing the maintenance costs.		
Alternatives	1. Leverage poor version of MVC that allows the view to talk to the model by the controller returning to the view reference to the model (e.g., allow the controller to return to the view Mutable Interface and/or Domain Object) **2. Leverage best practices version of MVC in which the view cannot have reference to the model at any point, thus preventing knowledge of the Mutable Interface and Domain Object.**		

Decision	Option 2 was selected, because it provides the more strict layering approach, decouples the view from the model reducing the impact of change, and simplifies the architecture by making the controller the only component that can talk to the model.

<table>
<tr><td>Subject Area</td><td>R4SC Service Component Pattern</td><td>AD ID#</td><td>App Area</td></tr>
<tr><td>Architectural Decision</td><td>Process Contracts should be designed for use by a single Mediator Function.</td><td>BC4.0</td><td>Business Controller</td></tr>
<tr><td>Problem Statement</td><td colspan="3">Should we create more generic, broader use contracts that can be leveraged by several Mediator Functions, or should we be specific about the data needs for a single service and limit the usability of the contract to a single Mediator Function?</td></tr>
<tr><td>Assumptions</td><td colspan="3">Optional information increases the complexity of the contract. Makes it more complicated to document and explain for use by integrators.
Optional information increases the likelihood of implementation errors, thus increasing the complexity of the contract validation that needs to be performed.
Case statements to support the business rules of optional information are very susceptible to change, and in many situations the case statements for different usage scenarios will diverge over time, making the logic more complicated and more difficult to maintain.
Optional information cannot be eliminated for all cases.
Very specific contracts that limit or eliminate the optional data reduce the potential for their reuse, thus increasing the number of classes that need to be maintained in the solution.
In the case of a shared Process Contract, when the data requirement needs of the Mediator Functions sharing the Process Contract diverge, there may be a need to split the Process Contract in two to support each masters unique needs. By eliminating this possibility by not sharing Process Contracts, the service signatures will be more stable over time, thus improving the maintainability of the application interfaces that leverage the services.</td></tr>
<tr><td>Motivation</td><td colspan="3">Simplify the usability of the Mediator Functions for the users that will have to design to the service API. Provide a solution that provides the most stable Process Mediator API for the service users to minimize the impact of change to the user throughout the life of the solution.</td></tr>
<tr><td>Alternatives</td><td colspan="3">Leverage a single Process Contract that contains accessor methods to all the information that will be needed by any Mediator Function in the system.
Share Process Contracts across Mediator Functions that have similar purposes to reduce the number of contracts that need to be maintained and take advantage of reuse.
Define a separate Process Contract for each Mediator Function on each Process Mediator.</td></tr>
<tr><td>Decision</td><td colspan="3">The Advantage of Option 1 is that it offers the fewest possible number of Process Contract classes to develop and maintain, one. The disadvantages are many… the contract implementation would be excessively large, which would be disadvantageous in a distributed environment. The contract would be complicated for users to design to, because the specific data needs for the service they need to leverage aren't obvious in the definition of the class. The documentation required to describe the data needs per Mediator Function would</td></tr>
</table>

	be difficult to develop, cumbersome to read through for the user, and hard to maintain as service needs change and as services are added. For these reasons option 1 was eliminated. Option 2 provides a means to reduce the number of contracts that would have to be developed and maintained by sharing contracts across Mediator Functions with similar needs. The cost of developing and maintaining a Process Contract, on the other hand, is nominal particularly when you consider that the Process Contract code will be generated 100% when forward engineering code within tools like Rational Rose, XDE, and RSA. Option 2 was not selected due to the risk that over time the information needs of even similar Mediator Functions could change; thus, requiring the contracts to be broken up, which would cause the Mediator Function signatures to change. The potential negative impact of this change effect to the various users of the Process Mediator are greater than the nominal benefits that will be reaped by reducing the number of contracts in the system through reuse of the contracts. **Option 3 was selected** because it provides the most straight forward definition of the information needs of a Mediator Function to the users of that service. Since the contract is used by only one Mediator Function, optional data will be minimized. It does have the disadvantage of requiring more contract classes to be developed and maintained, but via the generation of these components during forward engineering with Rational RSA, XDE, or Rose the cost is largely avoided. Option 3 also eliminates the possibility that over time a Mediator Function's signature will change due to it's contract type changing. Since only one Mediator Function uses a particular contract, the contract content can change to support the changing information needs of that Mediator Function. There is no risk that those information needs will diverge from another Mediator Function's needs, since no other Mediator Function leverages the contract. The risk and cost avoidance Option 3 offers over Options 2 and 1 make it a better long term solution.

Process Mediator and its' Mediator Function

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Mediator Functions must take as their only parameter an abstract implementation of the Interface pattern : a Process Contract	BC5.0	Business Controller
Problem Statement	How should information required by the Mediator Function be delivered?		
Assumptions	• Simplifying the interface between the requestor (view) and the model logic improves the communication between these teams, which has a positive affect on integration quality and the possibility of parallel development. • With regards to contracts, we commonly refer to the abstract implementation of the Interface Pattern as the Process Contract. • With regards to contracts, we commonly refer to the concrete implementation class as the Process Contract Impl.		
Motivation	Reduce maintenance costs by simplifying the architecture and decoupling the layers making them less susceptible to the impacts of change.		
Alternatives	1. Pass all information to the Mediator Function in a class that		

	encapsulates all the Process Mediator's information needs (e.g., Process Contract Impl). 2. **Pass all information to the Mediator Function in an abstract object that implements the Interface pattern, which encapsulates all the Process Mediator's information needs (e.g., Process Contract).** 3. Expose each piece of information needed by the Mediator Function as a separate parameter on the function signature.
Decision	Option 1 was not selected, because it uses a concrete class that is inflexible in its implementation as the vehicle for passing the required information. This solution doesn't allow for the requestor to determine how to implement the contract, which eliminates some alternatives that some of the requestors might need in order to optimize for different usage scenario (e.g., some users may need to enable the contract implementation for distributed usage for others this might be unnecessary overhead). Option 3 was not selected because it exposes all the information that the Mediator Function will use in the method signature. When the information needs of the Mediator Function change, the requestor will need to modify to the updated method signature. Implementation is complicated, as well, as the developer must confirm that the order of the attributes passed is consistent with the order specified in the method signature. But what about the situation when only a single piece of information is needed for the Mediator Function to do its' job? A Process Contract should still be created and passed to the Mediator Function for two reasons: 1. consistency, which will simplify the solution architecture, and 2. we repeatedly observe that as a solution ages many of the Mediator Functions that originally only required a single piece of information to do its' job require additional data inputs as the solution matures. **Option 2 was selected**, because it provides implementation flexibility for the requestors of the Mediator Function and encapsulates the Process Mediators data needs from the requestor. A Mediator Function should always define as its parameter type the Process Contract, rather than the Process Contract Impl.

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Mediator Functions can return one of five values: void, an Immutable Interface, a collection of Immutable Interfaces, a native type/Simple object (e.g., boolean or String), a collection of native types/Simple objects	BC6.0	Business Controller
Problem Statement	How must the information that the controller returns to requesters be limited to ensure that model-view-controller (MVC) is preserved?		
Assumptions	• R4SC Service Component Pattern leverages the MVC approach in which the controller is not allowed to return domain model references to the view.		
Motivation	Ensure consistency of return information from the Mediator Fucntion. Preserve the implementation of the MVC pattern.		
Alternatives	1. Leverage poor version of MVC that allows the controller to return model references to the view, thus allowing the Mediator Function to		

	return the Mutable Interface or Domain Object. **2. Leverage best practices version of MVC in which the controller cannot return to the view reference to the Domain Model at any point, thus preventing the return of the Mutable Interface and Domain Object.**
Decision	Option 1 does not keep the View separate from the Model as prescribed by the Model-View-Controller framework. This form of MVC has evolved out of the misconception that it will improve performance. The incremental gain in performance is countered by a significant degradation in maintainability and flexibility of the system. It also introduces added complexity to the solution architecture, because the view in some cases must go through the controller to access the model, while in other cases it can access the model directly. For these reasons, Option 1 was eliminated. Option 2 was selected as it employs the best practices approach to MVC, in which access to the view is strictly limited through the Controller. This limits the potential types of information that the Process Mediator can return to: void, an Immutable Interface, a collection of Immutable Interfaces, a native type or Simple object (e.g., boolean and String), or a collection of native types or Simple objects. Furthermore, null is not a valid return option.

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Mediator Function cannot pass reference to the Process Contract outside of the Process Mediator Layer	BC7.0	Business Controller
Problem Statement	Should a Mediator Function be allowed to pass the Process Contract to the Mutable Interface for use in the Domain Object components?		
Assumptions	• Objects that are passed across the layers of an architecture couple the layers, making the solution more susceptible to the impacts of change and therefore increase the total cost of ownership. • Process Contracts are designed for consumption by the Mediator Function and its' helper classes (e.g., A Contract Validator) to fulfill that service's information requirements.		
Motivation	Design an architecture that is flexible to changing business needs.		
Alternatives	1. Reuse the Process Contract by passing its' reference to the Mutable Interface for consumption by the Domain Model. **2. Extract the content from the Process Contract through the Mediator Function. Allow the Mediator Function to leverage this information to call the appropriate Mutable Interfaces to execute the request, passing the appropriate information to those Mutable Interfaces as needed. The Process Mediator drops reference to the Process Contract at the conclusion of the service method execution.**		
Decision	The primary driving force behind this architectural decision is coupling: how many classes know of another class in the architecture. As we pass the contract throughout the layers of the architecture, it becomes more externally coupled. As that coupling increases, so does the impact of change when that object is modified. If we allow an object from one layer of the architecture to be passed into the business and persistence layers we increase coupling. The purpose of strict layering is to reduce coupling. Taking the effort to architect the solution with the layers of R4SC, but then allowing the contract to be passed up and down		

	the layers is counter productive, significantly damanging the overall architecture of the solution. In addition to increasing the impact of change and total cost of ownership of the solution, doing this complicates the integration effort across the team. For these reasons, Option 1 was not chosen. Option 2 was chosen, because it ensures the greatest separation of concerns between the components of the Controller and Model. This will reduce the potential impact of change throughout the life of the solution, resulting in a reduced total cost of ownership. By reducing the dependencies between the Controller and Model, the delivery team can better work in parallel on these components accelerating the delivery timeframe and reducing the potential for integration defects. * the Mediator Function can still pass reference to the Process Contract to other Process Mediators and Contract Validators, as they are components of the Controller Layer.

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Mediator Functions talk to the Domain Objects through the Mutable Interfaces	BC8.0	Business Controller
Problem Statement	In several implementations, the possibility of reusing the process across organizations exists, but the process had to run on top of a different domain model. So, how do we build the Process Mediator so that it can be reused on top of a different Domain Object Models?		
Assumptions	• Solution must be able to provide benefits of decoupling without introducing noticeable performance overhead. • Possibility of leveraging the Mediator Function across domains in the future outweighs the cost of designing/implementing/maintaining the added separation of concern.		
Motivation	Decoupling the Mediator Function from the concrete implementation of the Domain Object reduces the impact of change and allows for greater flexibility in the implementation of the domain model. In layman's terms, since the Mediator Function doesn't know it is dealing with the Domain Object, the implementation of the Domain Object can change with little or no impact to the Mediator Function. Only in the event that the Mutable Interface must change will the Mediator Function need to be updated.		
Alternatives	1. Allow the Mediator Function to directly talk to the Domain Object. **2. Create an implementation of the Interface pattern that provides access to the modifier functions (setters) of the Domain Object, but allows the user to remain independent of the specific implementation.** 3. Add to the Immutable Interface to include the setter functions that we want to expose to the Mediator Function.		
Decision	Option 1 was not selected, because it does not allow the Mediator Function to be used on top of another domain model without significant rework. In the event, there is no perceived opportunity to use the Process Mediator on top of another Domain Model, it may still be advisable to separate the Process Mediator from knowledge of the Domain Model. Doing so will afford your Process Mediator greater resilience to changes in the Domain.		

	Option 3 was not selected, because the Immutable Interface is intended to be used as the return value of the Mediator Function to the integrator. Since the integrator is a part of the View (in terms of the MVC framework), we can not give it access to any features that would modify the state of the model (e.g., setter methods) or we would be in violation of the Model-View-Controller pattern. **Option 2 was chosen** because it provides a lightweight option for encapsulating the Mediator Function from the knowledge of the concrete Domain Model it sits upon, while not introducing a significant amount of effort to design/implement/maintain the additional layer. This lightweight approach to encapsulation provides for the reuse of the Process Mediators on top of a different domain, in the event that the Business Units require that flexibility. In the case where the Process Mediator will not need to be used with multiple Domain Models, this approach provides encapsulation benefits, which will reduce the impact of change in the Process Mediator when the Domain Model must change. By reducing the impact of change, we afford the business the opportunity to more easily change the Domain Model to meet changing business needs.

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	A Mediator Function can leverage other Mediator Functions to delivery its service.	BC9.0	Business Controller
Problem Statement	Should Mediator Functions be self-contained and depend on no external Mediator Function, or may a Mediator Function reuse services provided on other Process Mediators?		
Assumptions	• Objects in the same layer can have public access to one another without violating the best practices of object oriented architecture. • Object oriented best practices propose the reuse of functions rather than the duplicating the lines of code. • Duplicated lines of code increase complexity and hurt maintainability. • Reusing services from another object couple you to that object. As coupling increases between objects, so does the potential impact of change to the calling object when the called object changes.		
Motivation	Develop Mediator Fucntions that are easy to maintain and resilient to the impact of change.		
Alternatives	**1. Allow Mediator Functions to leverage other Mediator Functions.** 2. Don't allow Mediator Functions to leverage other Mediator Functions. 3. Allow Mediator Fucntions to leverage other Mediator Functions on it's Prcoess Mediator only.		
Decision	Option 2 was not selected, because it results in duplicating lines of code to perform the same function. This is inefficient from a development and maintenance standpoint, and poor OO design and implementation. It does provide the maximum decoupling among the Mediator Functions, but the consequences outweight the benefits of the reduced coupling. Option 3 was not selected for the same reason as option 2. Option 3 provides a little more flexibility in reusing services within the same Process Mediator, while retaining the same degree of decoupling. Since Option 3 prohibits the use of Mediator Functions on other Process Mediators, it would require the duplication of lines of code. Like Option 2, this is inefficient from a development and		

	maintenance standpoint, poor OO design, and increases total cost of ownership. **Option 1 was selected**, because it follows the recommended object oriented best practice of leveraging services within the same layer to improve development and maintenance efficiency, and reduce the complexity of the solution by isolating identical functionality to a shared service. Option 1 does increase the potential of coupling amongst the Mediators Functinos, which can have a negative effect on the impacts of change to the solution, however, the impact can be isolated within the Mediator Function itself in most cases. Therefore, the potential of the change impact reaching the requetors (the Provider Adapters), for example inter-layer coupling, of the service is no greater with option 1 than option 2 or 3.

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Process Mediators are intended to have many Mediator Functions	BC10.0	Business Controller
Problem Statement	What should a Process Mediator class contain? Should it have only a single Mediator Function?		
Assumptions	• Object metric best practices suggest that throughout a project, classes should have an average of 10-15 functions, with an average of 15 lines of code per function.		
Motivation	Develop Process Mediators that logically group services of the application to simplify the usage of the system and correlate with the real-world relationships.		
Alternatives	1. A Process Mediator class is defined for each Mediator Function. **2. A Process Mediator class consists of a group of related Mediator Functions.** 3. A Process Mediator class consists of Mediator Functions for a single Domain Object or hierarchy of Domain Objects.		
Decision	Option 1 was not selected, because it creates an unnecessarily large number of classes and does not result in groupings of like services as is prevalent in the business world. This option is less efficient from an implementation/maintenance standpoint, as it tends to increase the lines of code. Option 3 was not selected either. At first glance, one might ask "what is wrong with this option"? The intention of the Process Mediator is not to have a one to one correlation to the Domain Model. Doing so would tightly couple the Controller layer to the Model, which would have a negative impact on total cost of ownership. The intention of the Process Mediator is to encapsulate the primary business operations that are exposed to the user channels, and to make unrelated Domain Object work together to fulfill that business operation. **Option 2 was selected**, because it results in Process Mediators that are bundled similar to the real world business. It results in service oriented objects that focus on managing the business processes, by directing the Domain Objects to work together to deliver the services. This solution adheres better to the metric best practices for number of functions per object.		

Contract Validator

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	**Prohibit the duplication of rules in the Contract Validator**	BC11.0	Business Controller
Problem Statement	Should we allow enterprise business logic edits (e.g., legacy application edits) or database edits to be checked in the Contract Validator?		
Assumptions	• Significant performance improvements have not been observed by moving the Data edits closer to the UI. • Duplicating edits in the architecture results in maintenance consistency problems and greater number of defects.		
Motivation	The desire to identify business rule violations as quickly as possible in the system, results in the temptation to pull enterprise business rules and database edits closer to the user interface.		
Alternatives	1. Allow database edits (length edits, format edits, etc.) or enterprise business edits to be duplicated in the Contract Validator to detect errors earlier. **2. Prohibit the duplication of rules in the Contract Validator to avoid errors associated with keeping the duplicated rules in sync.**		
Decision	Option 1 provides for earlier detection for rules violations in the architecture; however it introduces unacceptable risk of errors due to rules becoming inconsistent, and it creates added cost in maintaining the solution. Duplicated rules are difficult to track and keep consistent, therefore introducing the possibility for defects. Additionally, placing database or enterprise application specific edits in other layers of the solution increases the coupling of the solution to those underlying information sources and reducing the flexibility. **Option 2 was selected** because it preventing the duplication of rules (e.g., database edits) and ensures the business logic of the application is decoupled from its underlying information sources (e.g., databases and enterprise apps). With the advent of good rules engines in the marketplace, there is now the possibility of adhering to Option 2, but achieving the earlier detection benefits sought in Option 1. By externalizing the rules to a Rules Engine, a solution's rules can remain unique (unduplicated), while being integrated into multiple different validation checkpoints. This approach does adhere to the architectural decision of Option 2, as it prevents duplication of rules.		

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Contact Validation should be managed by a support object that the Process Mediators leverage, rather than a function of the Process Mediators themselves.	BC12.0	Business Controller
Problem Statement	Should contract validation be a function of the Process Mediator or a separate support object within the application?		
Assumptions	• Process Mediators will have several Mediator Functions, which each requires a different Process Contract which may need validation. • Object oriented metrics best practices recommends that an application should have a class average of 10-15 methods per class, and 15 lines of code per method. Classes that greatly exceed these metrics should be reviewed		

	for the possibility of redesign.
Motivation	Provide a recommendation for designing The Contract Validator that is efficient to develop/maintain, allows for reuse of validation services, and does not result in excessively large or small classes.
Alternatives	1. Using method overloading, create a validate method on the Process Mediator itself for each Process Contract that needs to be validated. **2. Create a centralized Process Contract Validator object that contains a validate method for each Process Contract that needs to be validated for all the Mediator Functions in the solution. Leverage method overloading to ensure a consistent approach to accessing validation, thus simplifying the architecture.** 3. Create a Contract Validator object per Process Contract that needs to be validated. The object will provide a single validate method. 4. Define the validation of the Process Contract within the Process Contract itself.
Decision	This architectural decision doesn't have one right answer. Option 1, 2, and 3 are all acceptable and have their own inherent strengths and weaknesses. Of these option, option 2 is the preferred approach. When deciding on the approach to apply for a particular implementation, the architect must assess the strengths and weaknesses of each option and their relavence to the solution at hand. Option 1 has the benefit of keeping the validation encapsulated and private within the Process Mediator class. Its' weakness is, given the first assumption above, this option tends to result in Process Mediators that press the limits or exceed the 10-15 method best practice metrics of OO resulting in overly complex objects that are harder to maintain. **Option 2 is recommended**. By centrally locating the validation to a separate Contract Validator, we define the validation for a contract once and any Mediator Function can access the logic. Additionally, by removing the validation methods from the Process Mediator, we help to avoid the probability that the Process Mediators will exceed the method metrics suggestion of 10-15 methods per class. The benefit of accomplishing this is that we now have two types of very focused objects: Mediator Functions that focus on conducting various objects to deliver a business process, and validation objects that are focus exclusively on validation of contracts. The weaknesses of option 2 are that we will end up with a validation class with far more methods than the metric best practice of 10-15 in order to validate the entire applications suite of Process Contracts. Option 3 guarantees that the metrics best practice of 10-15 methods per class will not be exceeded, as was a concern of option 1 and 2. Option 3's weakness is that it results in a large number of classes, which increase as the numbers of Process Contracts increase. Extremely small objects, are inefficient to develop/maintain and should be candidates for redesign. Option 4 is not allowed, since the Process Contract is an implementation of the Interface pattern. The Interface pattern allows you to define the method signatures required, but does not allow you to define the method implementation. The method implementation is left open and flexible to the implementor of the Process Contract.

	What does that mean in layman's terms? It means that we could define a validate method on the Process Contract interface, but we could not enforce the actual rules that were applied in doing the validation by the implementor (implementor in this case means that actual class that implements the Process Contract interface). The end result, we would not be able to guarantee consistent rules applied in the validation process, across users of the Mediator Function, which defeats the purpose of the validation. Taking this thought process one step further, it has been recommended that we define the actual class that implements the Process Contract interface, so that we could define and enforce a consistent validate process. This approach would only work if we changed the Process Mediator service methods to require a Process Contract implementation class, rather than the Process Contract interface. That would violate Architectural Decision – ***BC4.0, Process Mediator Functions must take as their only parameter an abstract implementation of the Interface pattern : a Process Contract,*** defined in the Process Mediator's Architectural Decisions above.

Domain Factory

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Both the Domain Factory and Mutable Interface must be applied to encapsulate the controller from the Domain model.	BM10.0	Domain Model
Problem Statement	Allow for greater maintainability of the controller layer (Process Mediator, Mediator Functions) over time by eliminating direct knowledge of the Domain model.		
Assumptions	• The development/support team has the experience with design patterns, so that the benefits of maintainability will outweigh the cost of the additional complexity introduced by the indirection.		
Motivation	▪ Reduce the total cost of ownership and development of the solution. ▪ Allow for the reuse of the Process Components on top of multiple Domains.		
Alternatives	1. Apply the Domain Factory and Mutable Interface as a pair to encapsulate the Process Mediator from the Domain Model. 2. Leverage a Domain Factory to instantiate the Domain Objects, but do not require the Mutable Interface. 3. Leverage the Mutable Interface, but choose not to leverage the Domain Factory		
Decision	Option 2 was not chosen, because it does not address the motivation. The Domain Factory encapsulates the controller from the instantiation of the Domain Objects. Once the Domain Object is instantiated it must be returned to the controller for use. In order to achieve the encapsulation of the controller from the Domain Objects, the Domain Factory cannot return reference directly to the Domain Object itself. However, without a Mutable Interface the Domain Factory has no alternative. So, defining the Domain Factory without the Mutable Interface does not deliver the encapsulation of the controller from the Domain Objects. For that reason, Option 2 does not address the problem statement. **Option 1 was selected** because it encapsulates the controller from the Domain		

	Objects throughout the object lifecycle. The Domain Factory encapsulates the instantiation of the Domain Objects, while the Mutable Interface provides a generic specification for accessing the capabilities to change the state of the Domain Object without becoming coupled to the concrete implementation of the Domain Object. The Mutable Interface is returned by the Domain Factory to the controller to give it access to the domain, indirectly. In the event that the architect determines that the added indirection introduced by the Domain Factory is not appropriate given the teams' knowledge of Design Patterns, Option 3 (to leverage the Mutable Interface when not leveraging the Domain Factory) is a recommended alternative. Why? For one, there is a one-to-one relationship between the Mutable Interface and the Domain Object, so it does NOT introduce a great degree of indirection, like the Domain Factory. What it does do is encapsulate the Process Mediator from 95% of the potential coupling that would arise by directly communicating to the Domain Model. Therefore, the Mutable Interface greately reduces the potential impact of change on the Process Mediator with very little added complexity to the overall architecture.

Immutable Interface

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Interface will adhere to the guidelines of the Interface pattern and be implemented by it's Business Object	BM1.0	Model
Problem Statement	How can we create a view of the Business Model that can be returned by the Business Mediator Function to the Provider Router, but does not tie the Provider Router to a specific implementation of the Business Model?		
Assumptions	• Implementation cannot provide access to methods that would change the state of the Business Model (e.g., setter methods).		
Motivation	Architect a solution that preserves MVC in the most efficient means possible.		
Alternatives	1. Create an implementation of the Proxy pattern to be returned from the Business Mediator to the Provider Router. Data about the Business Object will be transferred to the lightweight, Proxy instance. **2. Create an implementation of the Interface pattern to be implemented by the Business Object. Allow the Interface to be returned by the Business Mediator Function to the Provider Router.**		
Decision	Option 1 provides a lightweight picture of the Business Object that can be beneficial when deployed on a distributed environment. Option 1 also increase the number of classes that need to be developed/maintained, and the number of instances that need to be instantiated and managed during run time. Typically, there is not a lot of extraneous data that is on the Business Object that would not be available for the Provider Router, so the Proxy itself is not much more lightweight than the original Business Object. Therefore, the cost of maintaining the additional classes and the performance degradation related to instantiating, populating, and garbage collecting the Proxy are not worth the lightweight benefits that might be delivered for a distributed application. **Option 2 was selected**. It provides better flexibility as the returned value is an Interface (an abstract definition of message signatures), rather than a concrete Business Object or Proxy. The interface actually represents a different view on		

	the Business Object, which means no additional objects need to be instantiated and populated, thus improving performance. While Option 1 was not selected as the preferred approach, there may be situations where it is superior to Option 2, such as distributed architecture situations where the payload sent across the network can be significantly reduced via a Proxy. Through performance modeling, architects should assess their particular situation to determine which alternative is best for their situation.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Interface can only expose getter methods that return native types, simple object types (e.g., String), or Immutable Interfaces	BM2.0	Model
Problem Statement	How can we ensure that the model-view-controller is preserved by minimizing the methods and the potential return types of the Immutable Interface to those that the View may have access to?		
Assumptions	• View cannot have access to any object that can modify the state of the Business Object (e.g, the Mutable Interface and Business Object). • View cannot have access to any object that can provide it access to an object that can modify the state of the Business Model (e.g, objects that can return Mutable Interfaces and Business Objects). • View receives information about the Business Model layer via the Immutable Interface.		
Motivation	Adherence to model-view-controller (MVC) will reduce impact of change to the implementation throughout the life of the application, thus reducing the maintenance costs.		
Alternatives	**1. Immutable Interfaces can provide getter methods that only return native types, simple object types (e.g., String), and other Immutable Interfaces.** 2. Immutable Interface can provide getter methods that return all types in option 1 plus Business Objects and Mutable Interfaces. 3. Immutable Interface can provide both getter and setter methods.		
Decision	**Option 1 was selected**, because it limits the methods that can be exposed on the Immutable Interface to those that do not change the state of the Business Object, as well as limit the return values from the methods to types that do not provide the ability to change the state of the Business Object. Option 2 was not selected, because though it limits the methods that can be exposed on the Immutable Interface to those that do not change the state of the Business Object, it does not limit the return values from the methods to types that do not provide the ability to change the state of the Business Object By allowing the getters to return Mutable Interfaces or Business Objects, the View will have access to change the state of the Business Model directly, which violates MVC. Option 3 was not selected, because it does allows the Immutable Interface to expose functions that would change the state of the Business Object, as well as return values that would allow the View access to change the state of the Business Objects.		

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	In the case that a Mediator Function is provided	BM3.0	Domain Model

	to initially retrieve a part of a Domain Object, the Domain Object's Immutable Interface should not expose the getter method for that part.		
Problem Statement	If a Mediator Function is provided to retrieve a part of a Domain Object, should the Immutable Interface provide a getter method to provide access to the same part?		
Assumptions	• Providing both a getter and a Mediator Function to the users to request the same information will cause confusion, misuse, and questions for the integrators. • Providing both requires case statements to assess whether to use the getter or Mediator Function in the Provider Adapter. Provider Adapter logic is likely to be unique per user, which results in redundant logic and introduces the possibility of inconsistency among user access channels.		
Motivation	Eliminate confusion as to when to call the Mediator Function and when to call the getter on the Immutable Interface. Eliminate complex case statements in the Provider Adapter to determine which should be used.		
Alternatives	1. Allow the Immutable Interface to provide access to a getter method for which there is a Mediator Function to retrieve the attribute. Require that the getter method raise an exception in the event that the attribute is currently lazy initialized and should be initialized through the Process Mediator. **2. Do not allow the Immutable Interface to provide access to the getter method. Force all retrieval of the attribute through the Process Mediator method. Expose the getter method on the Mutable Interface, so that the Process Mediator can determine if the attribute has yet been initialized. Getter method on the Mutable Interface should raise an exception indicating when the attribute has not yet been formally intialized and populated.** **If the attribute has not been initialized, the Process Mediator will trigger the process of retrieving the attribute and then return its' Immutable Interface to the Provider Adapter that triggered it. In the event that the attribute has been initialized (which would be signaled by the Immutable Interface not throwing its' exception) the Process Mediator would simply return the value as an Immutable Interface.**		
Decision	Option 1 was not chosen, because it is confusing to integrate for the implementation/maintenance team. More documentation is required to indicate to the user when they should go to the Process Mediator and when they should go to the Immutable Interface. In addition, Option 1 places more burden on each Provider Adapter that leverages the Process Mediator or Immutable Interface to manage this logic, resulting in an addition of lines of code to develop/maintain. **Option 2 was selected**, because it simplifies the usability of the Process Mediator/Immutable Interfaces that the View layer will interact with. Option 2 encapsulates the complex logic of initializing and returning the Immutable Interface data in these situations, thus requiring us to implement the logic once in the shared Mediator Function, rather than every time for each Provider Adapter that uses the Mediator Function (as would be the case in Option 1).		

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Immutable Interfaces getter methods should not	BM4.0	Domain Model

	have any parameters.		
Problem Statement	Should Immutable Interface getter methods be allowed to take parameter input data?		
Assumptions	• Parameter input data is typically used to trigger a state change or update the internal data of the Domain Objects. • Views, in the MVC sense, should not be allowed to make state or data changes to the Domain Model without going through the Controller (Process Mediator, Mediator Functions). • The Immutable Interface is NOT a part of the Controller layer. The Immutable Interface should not provide methods that may change the state of the Domain Objects. • Adhering to Model-View-Controller (MVC) is crucial for the maintainability and flexibility of the solution.		
Motivation	Design an architecture that adheres to MVC separation.		
Alternatives	1. Allow the getter methods of the Immutable Interface to have parameters in all situations. 2. Allow the getter methods of the Immutable Interface to have parameters in the situation where the parameter data is not used to trigger a state change in the Domain Model, nor used to update the data of the Domain Object. **3. Do not allow the getter methods of the Immutable Interface to have parameters.**		
Decision	Option 1 is not allowed, because it violates MVC. Option 1 would allow the user interface to change the state of the Domain Model without going through the Controller Layer (Process Mediator, Mediator Functions). Option 2 was not selected, because it introduces too many "what if" scenarios, therefore, complicating the decision tree of what is and isn't acceptable for the Immutable Interface. Goal is to define an architecture that is as user friendly as possilbe, and allowing Option 2 would not help us meet this objective. **Option 3 was selected**, because it ensures MVC adherence and does not introduce complex decision trees into the usability of the architecture. Rather than allow the Immutable Interface getter methods to take parameters, the parameters should be passed to Mediator Functions, via the Process Contracts.		

Mutable Interface

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Mutable Interface may provide access to public setter methods, service methods that change state, and getter methods that the designer did not want to expose to the View but must expose to the Process Mediator and other Domain Objects.	BM5.0	Model
Problem Statement	What methods can be exposed on the Mutable Interface?		
Assumptions	• There may be situations where a getter method is inappropriate to expose to the View, but is necessary to expose to the Process Mediator (Controller). • Only public services may be exposed through the Mutable Interface.		
Motivation	Define boundaries of the functions that a Mutable Interface may expose to its users.		
Alternatives	**1. Public setter methods.**		

	2. **Public getter methods that the user didn't want to expose to the User Interface through the Immutable Interface.** 3. **Public service methods that change the state of the Domain Objects other than the setter methods.** 4. Public getter methods that were exposed through the Immutable Interface. 5. Private or protected methods
Decision	Option 1, 2, and 3 are all approved services that may be defined on the Mutable Interface. Option 4 is not allowed because it is inefficient during development time. The Mutable Interface will inherit all of the getter methods defined on the Immutable Interface and should not redefine them on itself. Option 5 is not possible, because the Interface pattern only allows public methods.

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Mutable Interface methods can return native types, simple object types (e.g., String), Mutable Interfaces, Immutable Interfaces, or void.	BM6.0	Model
Problem Statement	What information can the Mutable Interfaces return to their users?		
Assumptions	• Users of the Mutable Interface can have access to objects that provide services that change the state of the Domain Model. • The Mutable Interface exists to minimize/eliminate coupling between the Process Mediator and the Domain Model, therefore, the Mutable Interface should not have any knowledge of a specific Domain Model.		
Motivation	Ensure that the users of the Mutable Interface do not get exposed to the Domain Objects themselves through the return value, thus preserving the separation of concern among the layers and objects.		
Alternatives	1. **Mutable Interfaces can provide methods that only return native types, simple object types (e.g., String), Mutable Interfaces, Immutable Interfaces and void.** 2. Mutable Interface can provide setter methods that return all types in option 1 and Domain Objects.		
Decision	**Option 1 was selected**, because it provides does not couple the Mutable Interface to a specific Domain, and ensures that its' users will remain decoupled from any knowledge of a specific Domain Model. Option 1 will provide greater insulation from changes in the Domain. Option 2 was not selected, because it violates the assumption that users of the Mutable Interface cannot have access to the Domain Objects. Option 2 would not guarantee us the potential to use the Process Mediator on top of another Domain Model, without changing the Mutable Interface itself. Changing the Mutable Interface would introduce significant change impacts to current users of the Mutable Interface.		

Domain Object

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Aggregate Domain Objects hold reference to the	DM7.0	Model

	Mutable Interface of each of its' parts		
Problem Statement	Object oriented solutions are founded on the concept of proper use of aggregation. However, even appropriate use of aggregation between Domain Objects increases the coupling in the system and therefore impacts its' flexibility and maintainability. How can we architect a solution that supports aggregation, but provides flexibility in its' implementation?		
Assumptions	• Aggregate whole, Domain Objects have a need to request the modification of the state of their parts.		
Motivation	Reducing the coupling between the specific implementations of the Domain Objects within a model will reduce the impact of change amongst aggregate whole Domain Objects when their parts change.		
Alternatives	1. Aggregate whole Domain Objects hold onto their parts' Domain Object reference. **2. Aggregate whole Domain Objects hold onto their parts' Mutable Interface reference.** 3. Aggregate whole Domain Objects hold onto their parts' Immutable Interface reference, and have the option at any time to cast that reference to it's Mutable Interface temporarily within the execution of a method.		
Decision	Option 1 was not chosen because it creates a high degree of coupling between the concrete implementation of the other Domain Objects in the Domain Model. This makes the solutions less flexible and more susceptible to the impacts of change. Option 2 greatly reduces the coupling between Domain Objects, therefore making the solution more maintainable and reducing total cost of ownership. Option 3, also greatly reduces the coupling between Domain Objects, however, This is accomplished in the case of Option 2 and 3, by leveraging a flexible, abstract definition of the parts (an Interface), rather than having direct knowledge of the concrete Domain Object. Option 3 creates added complexity for a Domain Object to affect a change of its aggregate parts. In the case of Option 2, the Domain Object has reference to its parts' Mutable Interface, and therefore has access to the methods that change the state of the parts. In the case of Option 3, the Domain Object has reference to its parts' Immutable Interface, which does not provide the methods to change the state of the parts. In order to change the state of an aggregate part in Option 3, the aggregate whole would first have to cast its Immutable Interface reference to the parts Mutable Interface, which would then provide it access to change the state of its' parts. Option 3 is inefficient and offers no greater decoupling benefit than option 2, **for that reason Option 2 was selected.**		

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Each Domain Object must implement it's corresponding Mutable Interface	DM8.0	Model
Problem Statement	If a Domain Object is not going to expose any setter functionality, why not have the Domain Object implement the Immutable Interface and not create a Mutable Interface for that Domain Object?		
Assumptions	• Consistent design and implementation simplifies a solution. • Over the lifecycle of any solution it is difficult to predict whether an object		

	will need to expose setter functionality, at some time. • Empty Mutable Interfaces add no implementation nor maintenance overhead.
Motivation	Simplify the solution through consistent implementation.
Alternatives	1. Domain Object implements it's Immutable Interface when it doesn't need to expose any state changing functionality. **2. Domain Object implements it's Mutable Interface.**
Decision	Option 1 was not chosen because it introduces inconsistency into the design and implementation. Since a majority of the Domain Objects do expose setter functionality at some point in the application, other Domain Objects are implementing their Mutable Interfaces and we should consistently adhere to that rule, even if the Mutable Interface is empty. As the solution matures, there is a good chance that the blank Mutable Interfaces will be populated to support future Process Mediators. **Option 2 was selected**.

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Instance creation and management of the Domain Object must be managed through a factory method.	DM9.0	Model
Problem Statement	Provide guidance for the instantiation and instance management of Domain Objects to best meet the needs of a solution throughout its lifespan.		
Assumptions	• Tailoring of instantiation strategies is necessary to address hot spots during performance testing. • It is difficult to anticipate your hot spots proactively, so a good architecture provides flexibility to make changes to address the hot spots reactively. • It is essential to be able to respond to these changes rapidly, as stress/performance testing is often in the critical path to delivery.		
Motivation	Accelerated time to market through flexibility. A solution that delivers cost savings and a positive ROI over the life of the project. We need to make sure that the solution is not more expensive to implement than the change impact that would be expienced with a less encapsulated solution. Over the life of a solution it often becomes necessary to switch instantiation strategies to meet the changing non-functional demands of a solution. The architecture should encapsulates, from the requestors, the knowledge of whether an instance-based, pooling, or singleton instantiation strategy is being used.		
Alternatives	1. Where an instance-based strategy is deemed appropriate, allow the Domain Object to expose the default instantiator as a public method to request instances. 2. Where a singleton strategy is deemed appropriate, allow the Domain Object to expose a singleton method. **3. Instance creation and management of the Domain Object must be managed through a factory method.**		
Decision	Option 1 locks the Domain Object into an instance-based approach. If, at some point in the future, it becomes necessary to transition to a singleton or pooled instantion strategy, the requestors of the Domain Object will need to change. In Java, this approach would have the requestor call the new() method on the Domain Object. Option 2 locks the Domain Object into a singleton strategy, resulting in the same		

	change impact as option 1 if a shift to an instance-based or pooling strategy is required. In this approach, a singleton operation (e.g., getSingleton) would be defined as a class operation on the Domain Object. The default instantiator would be made private to prevent anyone from creating new instances. An private attribute (singleton) would be defined on the Domain Object to hold reference to the singleton instance. The getSingleton operation would first check to see if the singleton attribute was populated. If it was it would return reference to that instance, if not populated, it would call the private default instantiator to populate the singleton attribute and return the reference to the instance to the caller. In option 3, you would define a factory method (e.g., getInstance), which would encapsulate knowledge of how the instances are being managed. This approach would require the the default instantiator (e.g., new() in Java) would be defined as a private method. If an instance-based strategy was needed for the Domain Object, the getInstance method would simply turn around and call the default instantiator returning the instance that was created. For a singleton situation, the getInstance method would check an attribute (e.g., singleton) to see if it was populated. If so, it would return the instance, if not it would call the default instantiator to create an instance, populate the singleton attribute with this instance, and return the reference to the singleton instance to the requestor. Finally, if a pooling scenario is required, the getInstance method would check an attribute that holds the collection of instances (e.g., instancePool) and “check out” one of the instances to return to the requestor. If all the instances were in use the getInstance method could, if allowed, request a new instance. The getInstance method would be responsible for managing the pool size as defined by the architectural decisions for the project. So, option 3 addresses the issue of encapsulation and provides a very flexible approach, thus addressing the first motivation. The second issue was whether this approach would be cost prohibitive, which this approach is not. The cost of designing and implementing this approach versus option 1 or 2 is nominal in comparison. For that reason, option 3 is required.

Architectural Decisions: Service Integration Pattern

Provider Router

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	A Provider Router function is limited to only routing messages to a single Process Mediator Function to ensure process logic does not get distributed across a variety of design elements.	PR 1.0	Process Router
Problem Statement	Should Provider Routers be able to coordinate many different Process Mediator Functions to fulfill the SOA Service that is being requested?		
Assumptions	• The Process Mediator/Process Mediator Functions are defined to control the logical unit of work, independent of the caller. • Within the architecture design framework, the most intuitive solution would limit the number of components that manage the service process to one.		
Motivation	Ensure that the process logic of the application is contained within as few components as possible, but not too few.		
Alternatives	1. **A Provider Router function is limited to only routing messages to a single Process Mediator Function to ensure process logic does not get distributed across a variety of design elements.** 2. Allow the Provider Router function to integrate multiple Process Mediator Functions in order to service the unique needs of the caller.		
Decision	If we allow the Provider Router to integrate multiple calls to Process Mediator Functions, we are effectively allowing the Provider Router to act as a composite for the Process Mediators. This results in the Provider Router, itself, being responsible for managing a process. The question then becomes could a Process Router be integrated by another Provider Router to make a larger grained operation? If we allow the Provider Router to manage a process to create a larger grained composite process, we now have two components in the architecture that have this responsibility: the Provider Router and the Process Mediator. This introduces confusion into the application of the architecture. When do I have the Process Mediator Function manage the process? What type of processes can the Provider Router manage? The purpose of the Process Router is to expose the Process Mediator Functions as SOA services, nothing more. To enforce that focused purpose, we do not allow for option 2. The Process Mediator Function is allowed to integreate other Process Mediator Functions in delivering its process (Architectural Decision BC9.0). If a composite is required to create a larger grained process, it is the Process Mediator Function's responsibility to implement this composite. Option 1, therefore, is the only allowed alternative.		

Architectural Decisions: Custom Application Development

Business Contract

Architectural Decisions:

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	A Business Contract encapsulates the information needs of the Business Mediator Function and is a specifications rather than concrete implementations	BC1.0	Business Controller
Problem Statement	How can requests for information that the Business Mediator Function requires be answered in order to perform its operation, while providing maximum flexibility in the implementation of those requests?		
Assumptions	▪ Business Mediator will be independent of the knowledge of those requesting its Business Mediator Functions. ▪ Implementation flexibility is critical to allow users to take advantage of various application distribution architectures and technologies.		
Motivation	Design a Business Mediator that can be leveraged across user channel technologies without a need to change the business logic implementation; thus, 1) ensuring consistency of response across user access points and 2) reducing maintenance by eliminating redundant silos of business logic per UI.		
Alternatives	1. Create a concrete class that encapsulates the requests for information and place an instance of this class as the parameter for the Business Mediator Functions. **2. Leverage the interface pattern to create an abstract definition of the request for information that the Business Mediator Function expects answered. Place an instance of this interface as the parameter for the Business Mediator Function.**		
Decision	Option 2 was selected, because it allows for implementation flexibility, thus better addressing the second assumption.		

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	Business Contracts are only allowed to contain getter methods and inquiry methods (e.g., “is…”)	BC2.0	Business Controller
Problem Statement	Should the contract be used as a uni-directional transport of information to the Business Mediator Function or should the contract be allowed to be a bi-directional transport of information between the Business Mediator and its requestor?		
Assumptions	• The standard protocol for returning information in OO languages is to use the return value to return requested information from a service to its requestor.		
Motivation	A goal of the Business Contract Layer is to provide a simple, user friendly API for users of the Business Mediator. The solution should require as little documentation as possible for a user to understand the appropriate way to integrate it. The degree to which unique details per instance of the contract can be reduced/eliminated adds to the simplicity of the solution.		
Alternatives	1. Allow the Business Contract to contain setter methods, thus allowing the Business Mediator Function to add information gathered during its execution.		

	2. Allow the Business Contract to provide only getter methods, thus reducing the contract to a uni-direction vessel for passing information to the Business Mediator Function for the operation's consumption.
Decision	Option 1 was not selected, because it complicates the usability of the contract. This approach results in contracts with getter and setter methods defined for two different users: the Business Mediator Function and the requestor. What information must the requestor provide to the Business Mediator Function for it to operate successfully? Which of the getters is provided for the requestor to receive information? When can the requestor expect to be able to call the getter methods whose data is populated by the Business Mediator Function and have valid information? **Option 2 was selected**, because it provides the most user friendly alternative. The contract has only one user: the Business Mediator Function. All the getter methods on the Business Contract are expected by the Business Mediator Function, and therefore, must be provided by the requestor (no question as to who provides information and when it needs to be available). Forces the Business Mediator Function to use the return value to provide information back to the requestor, which is a better practice than placing return information on the contract. Additionally, Option 2 ensures that the Business Mediator Function does not have the access to change the internal state of the Business Contract, thus preventing it changing the requestor's provided input.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Business Contract methods can return native types, simple object types (e.g., String), and Immutable Interfaces	BC3.0	Business Controller
Problem Statement	How can the potential return types of the contract that the requestor will have access to be minimized so that the model-view-controller (MVC) principles are preserved?		
Assumptions	• Users of the Business Mediator Function will not have access to Mutable Interfaces or BusinessObjects		
Motivation	Adherence to model-view-controller (MVC) will reduce impact of change to the implementation throughout the life of the application, thus reducing the maintenance costs.		
Alternatives	1. Leverage poor version of MVC that allows the view to talk to the model once the model is handed back to the view by the controller, thus allowing for communication to the Mutable Interface and Business Object **2. Leverage best practices version of MVC in which the view cannot have reference to the business model at any point, thus preventing knowledge of the Mutable Interface and Business Object.**		
Decision	Option 2 was selected, because it provides the stricter layering approach, decouples the view from the model reducing the impact of change, and simplifies the architecture by making the controller the only component that can talk to the model.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Business Contracts should be designed for use by a single Business Mediator Function.	BC4.0	Business Controller

Problem Statement	Should we create more generic, broader contracts that can be leveraged by several Business Mediator Functions, or should we be specific about the data needs for a single operation and limit the usability of the contract to a single Business Mediator Function?
Assumptions	• Optional information increases the complexity of the contract. Makes it more complicated to document and explain for use by end users. • Optional information increases the likelihood of implementation errors, thus increasing the complexity of the contract validation that needs to be performed. • Case statements to support the business rules of optional information are very susceptible to change, and in many situations the case statements for different usage scenarios will diverge over time, making the logic more complicated and more difficult to maintain. • Optional information cannot be eliminated for all cases. • Very specific contracts that limit or eliminate the optional data reduce the potential for their reuse, thus increasing the number of classes that need to be maintained in the solution. • Unique contracts per Business Mediator Function eliminate the possibility of a shared contract having to be split into two in the future when the information rules for the various Business Mediator Functions using the contract diverge. By eliminating this possibility, the Business Mediator Functions' signatures will be more stable over time, thus improving the maintainability of the requestors of the Business Mediator Functions.
Motivation	Simplify the usability of the Business Mediator Functions for the requestors integrating them. Provide a solution that provides the most stable Business Mediator Function signature for the requestors to minimize the impact of change throughout the life of the solution.
Alternatives	1. Leverage a single Business Contract that contains accessor methods to all the information that will be needed by any Business Mediator Function in the system. 2. Share Business Contracts across Business Mediator Functions that have similar purposes to reduce the number of contracts that need to be maintained and take advantage of reuse. **3. Define a Business Contract for each Business Mediator Function.**
Decision	Option 1 was not selected for several reasons: the contract implementation would be excessively large, which would be disadvantageous in a distributed environment. The contract would be complicated for requestors to design to, because the specific data needs for their desired Business Mediator Function aren't obvious in the definition of an all encompassing Business Contract. The documentation required to describe the data needs per Business Mediator Function would be difficult to develop, cumbersome to read through for the requestor, and hard to maintain as Business Mediator Function needs change and as Business Mediator Functions are added. Option 2 was not selected due to the risk that over time the information needs of even similar Business Mediator Function could change; thus, requiring the contracts to be broken up, which would cause the Business Mediator Function signatures to change. The potential negative impact of this change effect to the various users of the Business Mediator Function are greater than the benefits that will be reaped by reducing the number of contracts in the system through reuse of the contracts.

	Option 3 was selected because it provides the most straight forward definition of the information needs of a Business Mediator Function to the requestor of that service. Since the contract is used by only one Business Mediator Function, optional data will be significantly reduced. Option 3 also eliminates the possibility that over time a Business Mediator Function's signature will change due to it's contract type changing. Since only one Business Mediator Function uses a particular contract, the contract content can change to support the changing information needs of that Business Mediator Function. There is no risk that those information needs will diverge from another Business Mediator Function's needs, since no other Business Mediator Function leverages the contract.

Business Mediator/Business Mediator Function

Architectural Decisions:

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Business Mediator Functions must take as their only parameter an abstract implementation of the Interface pattern : a Business Contract	BC5.0	Business Controller
Problem Statement	How should information required by the Business Mediator Function be passed to the business operation?		
Assumptions	• Simplifying the interface between the UI and the business logic improves the communication between these teams, which has a positive affect on integration quality and the possibility of parallel development. • With regards to contracts, we commonly refer to the abstract implementation of the Interface Pattern as the Business Contract. • With regards to contracts, we commonly refer to the concrete implementation class as the Business Contract Implementation.		
Motivation	Reduce maintenance costs by simplifying the architecture and decoupling the layers making them less susceptible to the impacts of change.		
Alternatives	1. Pass all information to the Business Mediator Function in a class that encapsulates all the Business Mediator Function's information needs (e.g., Business Contract Impl). **2. Pass all information to the Business Mediator Function in an abstract object that adheres to the Interface pattern, which encapsulates all the Business Mediator Function's information needs (e.g., Business Contract).** 3. Expose each piece of information needed by the Business Mediator Function as a separate parameter on the Business Mediator Function signature.		
Decision	Option 1 was not selected, because it uses a concrete class that is inflexible in its implementation as the vehicle for passing the required information. This solution doesn't allow for the requestor to determine how to implement the contract, which eliminates some alternatives that some of the requestors might need in order to optimize the Business Mediator Functions usage (e.g., some users may need to enable the contract implementation for distributed usage, for others this might be unnecessary overhead). Option 3 was not selected because it exposes all the information that the Business Mediator Function will use in the Business Mediator Function signature. When the information needs of the Business Mediator Function		

	change, the requestor will need to change to the updated method signature. Implementation is complicated, as well, as the developer must confirm that the order of the attributes passed is consistent with the order specified in the Business Mediator Function signature. **Option 2 was selected**, because it provides implementation flexibility for the requestors of the Business Mediator Function and encapsulates the Business Mediator Function's data needs from the requestor. Business Mediator Functions should always define the Business Contract as its parameter type, rather than the Business Contract Impl.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Business Mediator Functions can return one of five values: void, an Immutable Interface, a collection of Immutable Interfaces, a native type/Simple object (e.g., boolean or String), a collection of native types/Simple objects	BC6.0	Business Controller
Problem Statement	How can the information that the controller can return to requesters be limited to ensure that model-view-controller (MVC) is preserved?		
Assumptions	• R4SC leverages the MVC approach in which the controller is not allowed to return business model references to the view.		
Motivation	Ensure consistency of return information from the Business Mediator Function. Preserve the implementation of the MVC pattern.		
Alternatives	1. Leverage poor version of MVC that allows the controller to return model references to the view, thus allowing the Business Mediator Function to return the Mutable Interface or Business Object. **2. Leverage best practices version of MVC in which the controller cannot return to the view reference to the business model at any point, thus preventing the return of the Mutable Interface and Business Object.**		
Decision	Option 2 was selected as it employees the best practices approach to MVC, which means the only types of information that the Business Mediator Function can return are: void, an Immutable Interface, a collection of Immutable Interfaces, a native type or Simple object (e.g., boolean and String), or a collection of native types or Simple objects. Furthermore, null is not a valid return option.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Business Mediator Functions cannot pass reference to the Business Contract outside of the Business Controller Layer	BC7.0	Business Controller
Problem Statement	Should the Business Mediator Function be allowed to pass the Business Contract to the Mutable Interface for use with the Integration Mediator? Content of Business Contract and Integration Contract are sometimes similar.		
Assumptions	• Objects that are passed across the layers of an architecture couple the layers, making the solution more susceptible to the impacts of change and therefore increase the total cost of ownership. • Business Contracts are designed for consumption by the Business Mediator		

	Function and its' helper classes (e.g., Contract Validator) within the Business Controller Layer to fulfill that service's information requirements.
Motivation	Design an architecture that is flexible to changing business needs.
Alternatives	1. Reuse the Business Contract by passing its' reference to the Mutable Interface, which may then forward the contract's reference to the Integration Mediator. **2. Extract the content from the Business Contract through the Business Mediator Function. Allow the Business Mediator Function to leverage this information to call the appropriate Mutable Interfaces to execute the request, passing the appropriate information to those Mutable Interfaces as needed. The Business Mediator Function drops reference to the Business Contract at the conclusion of its execution.**
Decision	Option 2 was chosen, because it adheres to the MVC pattern and best practices of strict layering. Both of which promote and deliver solutions that reduce the impacts of change. The general rule of thumb is coupling: how many classes know of another class in the architecture. As we pass the contract throughout the layers of the architecture, it becomes more externally coupled. As coupling increases, so does the impact of change when that object is modified. If we allow an object from one layer of the architecture to be passed into the business and persistence layers, we slowly dissolve the layering and then the change impact affects the entire application. * the Business Mediator Function can still pass reference to the Business Contract to other components of the Business Controller Layer (e.g., Contract Validator).

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Business Mediator Functions talks to the Business Object through the Mutable Interface	BC8.0	Business Controller
Problem Statement	In several implementations, the possibility of reusing the business process across organizations existed, but the business process had to run on top of a different business domain. So, how do we build the Business Mediator so that it can be reused on top of a different Business Model?		
Assumptions	• Solution must be able to provide benefits of decoupling without introducing noticeable performance overhead. • Possibility of leveraging the Business Mediator across business domains in the future outweighs the cost of designing/implementing/maintaining the Mutable Interface components.		
Motivation	Decoupling the Business Mediator from the concrete implementation of the Business Model reduces the impact of change and allows for greater flexibility in the implementation of the Business Model. In layman's terms, since the Business Mediator doesn't know it is dealing with the Business Model, the implementation of the Business Model can change with little or no impact to the Business Mediator. Only in the event that the Mutable Interface must change will the Business Mediator need to be updated.		
Alternatives	1. Allow the Business Mediator to directly talk to the Business Model. **2. Define an entity that adheres to the definition of the Interface pattern**		

	that provides access to the modifier functions (setters) of the Business Model, but allows the user to remain independent of the specific implementation. 3. Add to the Immutable Interface to include the setter functions that we want to expose to the Business Mediator.
Decision	Option 1 was not selected, because it does not allow the Business Mediator to be used on top of another Business Model without significant rework. Option 3 was not selected, because the Immutable Interface is intended to be used as the return value of the Business Mediator Function to the Provider Router. Since the Provider Router is a part of the View, we can not give it access to any features that would modify the state of the Business Model (e.g., setter methods) or we would be in violation of the Model-View-Controller framework. **Option 2 was chosen** because it provides the flexibility to reuse the Business Mediator on top of another Business Model, while not introducing a significant amount of effort to design/implement/maintain the additional layer. In the event that the implementation language does not allow for the efficient execution of the Interface pattern, thus introducing performance overhead during runtime, Option 1 may be used as an alternative. As an example, in the Java programming language, the execution of the interface pattern is extremely efficient as it does not require additional messaging or instantiation of objects. Instead, the Business Object actually is the Mutable Interface and all requests to the Mutable Interface are instantly forwarded by the compiler as calls to the Business Object. So, when using the Java language Option 2 should always be leveraged.

<table>
<tr><th>Subject Area</th><th>R4SC</th><th>AD ID#</th><th>App Area</th></tr>
<tr><td>Architectural Decision</td><td>A Business Mediator Function can leverage other Business Mediator Functions to delivery its operation.</td><td>BC9.0</td><td>Business Controller</td></tr>
<tr><td>Problem Statement</td><td colspan="3">Should Business Mediator Functions be self-contained and depend on no external Business Mediator Functions, or may a Business Mediator Function reuse other Business Mediator Functions?</td></tr>
<tr><td>Assumptions</td><td colspan="3">• Objects in the same layer can have public access to one another without violating the best practices of layered architecture.
• Object oriented best practices propose the reuse of functions rather than the duplicating the lines of code.
• Duplicated lines of code increase complexity and hurt maintainability.
• Reusing services from another object couple you to that object. As coupling increases between objects, so does the potential impact of change to the calling object.</td></tr>
<tr><td>Motivation</td><td colspan="3">Develop Business Mediators that are easy to maintain and resilient to the impact of change.</td></tr>
<tr><td>Alternatives</td><td colspan="3">1. Allow Business Mediator Functions to leverage other Business Mediator Services Functions.
2. Don’t allow Business Mediator Functions to leverage other Business Mediator Functions, rather have the Business Mediator Functions be self-</td></tr>
</table>

	contained. 3. Allow Business Mediator Fucntions to leverage other Business Mediator Functions on its Business Mediator only.
Decision	Option 2 was not selected, because it results in duplicating lines of code to perform the same function. This is inefficient from a development and maintenance standpoint, and poor OO design and implementation. It does provide the maximum decoupling among the Business Mediator, but the expense is too great. Option 3 was not selected for the same reason as option 2. Option 3 provides a little more flexibility in reusing Business Mediator Functions, while retaining the same degree of decoupling. But the expense to the development and maintenance are too great. **Option 1 was selected**, because it follows the recommended object oriented best practice of leveraging other objects and their operations within the same layer to improve development and maintenance efficiency, and reduce the complexity of the solution by isolating identical functionality to a shared operation. Option 1 does increase the potential of coupling amongst the Business Mediators, which can have a negative effect on the impacts of change to the solution, however, the impact can be isolated within the Business Mediator itself in most cases. Therefore, the potential of the change impact reaching the users (the Provider Routers), i.e. inter-layer coupling, of the service is no greater with option 1 than option 2 or 3.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Business Mediators are not intended to have a single Business Mediator Function	BC10.0	Business Controller
Problem Statement	What should a Business Mediator class contain? Should it have only a single Business Mediator Function?		
Assumptions	• Object metric best practices suggest that throughout a project, classes should have an average of 10-15 methods, with an average of 15 lines of code per method.		
Motivation	Develop Business Mediators that logically group Business Mediator Functions of the application to simplify the usage of the system and correlate with the real-world business operation relationships.		
Alternatives	1. A Business Mediator class is defined for each Business Mediator Function. **2. A Business Mediator class consists of a group of related Business Mediator Functions from a business perspective.** 3. A Business Mediator class consists of Business Mediators Functions for a single Business Object or hierarchy of Business Objects.		
Decision	Option 1 was not selected, because it creates an unnecessarily large number of classes and does not result in groupings of like services as is prevalent in the business world. This option is less efficient from an implementation/maintenance standpoint, as it tends to increase the lines of code. Option 3 was not selected either. At first glance, one might ask "what is wrong with this option"? The answer, Business Mediators should be looked at as groupings of like "business" operations, rather than grouping of services for a business object. The concern with this option is that it may lead to the Business Mediator doing nothing more than method chaining to the Business Model,		

	which results in business process information transferring to the Business Model. As this process information transfers, the coupling within the Business Model will increase, which will make it more brittle. **Option 2 was selected**, because it results in Business Mediators that are bundled similar to the way the business views their primary processes being bundled. It results in service oriented objects that focus on managing the processes and directing the objects to work together to deliver the services. It does not result in a Business Mediator per Business Mediator Function or a single Business Mediator with all the Business Mediator Functions for a Business Object.

Contract Validator

Architectural Decisions:

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	**Prohibit the duplication of rules in the Contract Validator**	BC11.0	Business Controller
Problem Statement	Should we allow enterprise business logic edits (e.g., operational system edits) or database edits to be checked in the Contract Validator?		
Assumptions	• Significant performance improvements have not been observed by moving the Data edits closer to the View. • Duplicating edits in the architecture results in maintenance consistency problems and greater number of defects.		
Motivation	The desire to identify business rule violations as quickly as possible in the system, results in the temptation to pull operational system rules and database edits closer to the View.		
Alternatives	1. Allow database edits (length edits, format edits, etc.) or operational system edits to be duplicated in the Contract Validator to detect errors earlier. **2. Prohibit the duplication of rules in the Contract Validator to avoid errors associated with keeping the duplicated rules in sync.**		
Decision	Option 1 provides for earlier detection for rules violations in the architecture; however, it introduces unacceptable risk of errors due to rules becoming inconsistent, and it creates added cost in maintaining the solution. Duplicated rules are difficult to track and keep consistent, therefore introducing the possibility for defects. Additionally, placing database or operation system specific edits in other layers of the solution increases the coupling of the solution to those underlying integration points and reducing the flexibility of the overall solution. **Option 2 was selected** because it preventing the duplication of rules (e.g., database edits) and ensures the business logic of the application is decoupled from its underlying integration sources (e.g., databases, operational systems, integrated SOA services). With the advent of good rules engines in the marketplace, there is now the possibility of adhering to Option 2, but achieving the earlier detection benefits		

	sought in Option 1. By externalizing the rules to a Rules Engine, a solution's rules can remain unique (unduplicated), while being integrated into multiple different validation checkpoints. This approach does adhere to the architectural decision of Option 2, as it prevents duplication of rules.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Contract Validation should be managed by a support object that the Business Mediator Functions leverage, rather than a function of the Business Mediators themselves.	BC12.0	Business Controller
Problem Statement	Should contract validation be a function of the Business Mediators or a separate support object within the application?		
Assumptions	• Business Mediators may have several Business Mediator Functions , which each require a different Business Contract. Each of those contracts may need to be validated. • Object oriented metrics best practices recommends that a application should have a class average of 10-15 methods per class, and 15 lines of code per method. Classes that greatly exceed these metrics should be reviewed for the possibility of redesign.		
Motivation	Provide a design recommendation for the Contract Validator that is efficient to develop/maintain, allows for reuse of validation services, and does not result in excessively large or small classes.		
Alternatives	1. Using method overloading, create a validate method on the Business Mediator itself for each Business Contract that needs to be validated. **2. Create a centralized Contract Validator that contains a validate method for each Business Contract that needs to be validated for all the Business Mediators in the solution. Leverage method overloading to ensure a consistent approach to accessing validation, thus simplifying the architecture.** 3. Create a Contract Validator per Business Contract that needs to be validated. The object will provide a single validate method. 4. Define the validation of the Business Contract within the Business Contract itself.		
Decision	This architectural decision doesn't have one right answer. Option 1, 2, and 3 are all acceptable and have their own inherent strengths and weaknesses. Of these option, option 2 is the preferred approach. When deciding on the approach to apply for a particular implementation, the architect must assess the strengths and weaknesses of each option and their relavence to the solution at hand. Option 1 has the benefit of keeping the validation encapsulated and private within the Business Mediator class. Its' weakness is, given the first assumption above, this option tends to result in Business Mediators that press the limits or exceed the 10-15 method best practice metrics of OO resulting in overly complex objects that are harder to maintain; **Option 2 is recommended**. By centrally locating the validation to a separate Contract Validator, we define the validation for a contract once and any Business Mediator Function can access the logic. Additionally, by removing the validation methods from the Business Mediator, we help to avoid the probability that the Business Mediators will exceed the method metrics suggestion of 10-15 methods per class. The benefit of accomplishing this is that we now have two		

	types of focused objects: Business Mediators that focus on coordinating various objects to deliver a business process, and validation objects that are focused exclusively on validation of contracts. The weakness of option 2 is that we will end up with a validation class with far more methods than the metric best practice of 10-15 in order to validate the entire applications suite of Business Contracts. Option 3 guarantees that the metrics best practice of 10-15 methods per class will not be exceeded, as was a concern of option 1 and 2. Option 3's weakness is that it results in a large number of classes which grow exponentially in numbers as the Business Contracts increase in number. Extremely small objects are inefficient to develop/maintain and should be candidates for redesign. <u>Option 4 is not allowed</u>, since the Business Contract is an implementation of the Interface pattern. The Interface pattern allows you to define the method signatures required, but does not allow you to define the method implementation. The method implementation is left open and flexible to the implementor of the Business Contract. What does that mean in layman's terms? It means that we could define a validate method on the Business Contract interface, but we could not enforce the actual rules that were applied in doing the validation by the implementor (implementor in this case means the class that implements the Business Contract interface). The end result, we would not be able to guarantee consistent rules applied in the validation process, across users of the Business Mediator Function, which defeats the purpose of the validation. Taking this thought process one step further, it has been recommended that we define the actual class that implements the Business Contract interface, so that we could define and enforce a consistent validate process. This approach would only work if we changed the Business Mediator Function to require a Business Contract implementation class, rather than the Business Contract interface. That would violate Architectural Decision – ***BC5.0, Business Mediator Functions must take as their only parameter an abstract implementation of the Interface pattern: a Business Contract,*** defined in the Business Mediator's Architectural Decisions above.

Immutable Interface

<u>Architectural Decisions:</u>

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Interface will adhere to the guidelines of the Interface pattern and be implemented by it's Business Object	BM1.0	Model
Problem Statement	How can we create a view of the Business Model that can be returned by the Business Mediator Function to the Provider Router, but does not tie the Provider Router to a specific implementation of the Business Model?		
Assumptions	• Implementation cannot provide access to methods that would change the state of the Business Model (e.g., setter methods).		
Motivation	Architect a solution that preserves MVC in the most efficient means possible.		
Alternatives	1. Create an implementation of the Proxy pattern to be returned from the		

	Business Mediator to the Provider Router. Data about the Business Object will be transferred to the lightweight, Proxy instance. **2. Create an implementation of the Interface pattern to be implemented by the Business Object. Allow the Interface to be returned by the Business Mediator Function to the Provider Router.**
Decision	Option 1 provides a lightweight picture of the Business Object that can be beneficial when deployed on a distributed environment. Option 1 also increase the number of classes that need to be developed/maintained, and the number of instances that need to be instantiated and managed during run time. Typically, there is not a lot of extraneous data that is on the Business Object that would not be available for the Provider Router, so the Proxy itself is not much more lightweight than the original Business Object. Therefore, the cost of maintaining the additional classes and the performance degradation related to instantiating, populating, and garbage collecting the Proxy are not worth the lightweight benefits that might be delivered for a distributed application. **Option 2 was selected**. It provides better flexibility as the returned value is an Interface (an abstract definition of message signatures), rather than a concrete Business Object or Proxy. The interface actually represents a different view on the Business Object, which means no additional objects need to be instantiated and populated, thus improving performance. While Option 1 was not selected as the preferred approach, there may be situations where it is superior to Option 2, such as distributed architecture situations where the payload sent across the network can be significantly reduced via a Proxy. Through performance modeling, architects should assess their particular situation to determine which alternative is best for their situation.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Interface can only expose getter methods that return native types, simple object types (e.g., String), or Immutable Interfaces	BM2.0	Model
Problem Statement	How can we ensure that the model-view-controller is preserved by minimizing the methods and the potential return types of the Immutable Interface to those that the View may have access to?		
Assumptions	• View cannot have access to any object that can modify the state of the Business Object (e.g, the Mutable Interface and Business Object). • View cannot have access to any object that can provide it access to an object that can modify the state of the Business Model (e.g, objects that can return Mutable Interfaces and Business Objects). • View receives information about the Business Model layer via the Immutable Interface.		
Motivation	Adherence to model-view-controller (MVC) will reduce impact of change to the implementation throughout the life of the application, thus reducing the maintenance costs.		
Alternatives	**1. Immutable Interfaces can provide getter methods that only return native types, simple object types (e.g., String), and other Immutable Interfaces.** 2. Immutable Interface can provide getter methods that return all types in option 1 plus Business Objects and Mutable Interfaces.		

	3. Immutable Interface can provide both getter and setter methods.
Decision	**Option 1 was selected**, because it limits the methods that can be exposed on the Immutable Interface to those that do not change the state of the Business Object, as well as limit the return values from the methods to types that do not provide the ability to change the state of the Business Object. Option 2 was not selected, because though it limits the methods that can be exposed on the Immutable Interface to those that do not change the state of the Business Object, it does not limit the return values from the methods to types that do not provide the ability to change the state of the Business Object By allowing the getters to return Mutable Interfaces or Business Objects, the View will have access to change the state of the Business Model directly, which violates MVC. Option 3 was not selected, because it does allows the Immutable Interface to expose functions that would change the state of the Business Object, as well as return values that would allow the View access to change the state of the Business Objects.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	In the case that a Business Mediator Function is provided to initially retrieve a part of a Business Object, the Business Object's Immutable Interface should not expose the getter method for that part.	BM3.0	Business Model
Problem Statement	If a Business Mediator Function is provided to retrieve a part of a Business Object, should the Immutable Interface provide a getter method to provide access to the same part?		
Assumptions	• Providing both a getter and a Business Mediator Function to the users to request the same information will cause confusion, misuse, and questions. • Providing both requires case statements to assess whether to use the getter or Business Mediator Function in the Provider Router. Provider Router logic is likely to be unique per user, which results in redundant logic and introduces the possibility of inconsistency among user access channels.		
Motivation	Eliminate confusion as to when to call the Business Mediator Function and when to call the getter on the Immutable Interface. Eliminate complex case statements in the Provider Router to determine which should be used.		
Alternatives	1. Allow the Immutable Interface to provide access to a getter method for which there is a Business Mediator Function to retrieve the attribute that the getter method provides access. But, require that the getter method raise an exception in the event that the attribute is currently lazy initialized and should be initialized through the Business Mediator Function. **2. Do not allow the Immutable Interface to provide access to the getter method. Force all retrieval of the attribute through the Business Mediator Function. Expose the getter method on the Mutable Interface, so that the Business Mediator Function can determine if the attribute has yet been initialized. Getter method on the Mutable Interface should raise an exception indicating when the attribute has not yet been formally intialized and populated.** **If the attribute has not been initialized, the Business Mediator Function will trigger the process of retrieving the attribute from persistence and**		

	then return its' Immutable Interface to the Provider Router that triggered it. In the event that the attribute has been initialized (which would be signaled by the Immutable Interface not throwing its' exception) the Business Mediator Function would simply return the value as an Immutable Interface.
Decision	Option 1 was not chosen, because it is confusing to integrate for the implementation/maintenance team. More documentation is required to indicate to the requestor when they should go to the Business Mediator Function and when they should go to the Immutable Interface. In addition, Option 1 places more burden on each Provider Router that leverages the Business Mediator Function or Immutable Interface to manage this complex logic, resulting in additional lines of code to develop/maintain. **Option 2 was selected**, because it simplifies the usability of the Business Mediator Function/Immutable Interfaces that the Consumer layer will interact with. Option 2 encapsulates the complex logic of initializing and returning the Immutable Interface data in these situations, thus requiring us to implement the logic once in the shared Business Mediator Function, rather than every time for each Provider Router that uses the Business Mediator Function (as would be the case in Option 1).

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Interfaces getter methods should not have any parameters.	BM4.0	Business Model
Problem Statement	Should Immutable Interface getter methods be allowed to take parameter input data?		
Assumptions	• Parameter input data is typically used to trigger a state change or update the internal data of the Business Model. • Access Channel should not be allowed to make state or data changes to the Business Model without going through the business controller layer (Business Mediator). • The Immutable Interface is NOT a part of the Controller layer, which the View can use to update the state or internal data of the Business Model. • Adhering to Model-View-Controller (MVC) is crucial for the maintainability and flexibility of the solution.		
Motivation	Design an architecture that adheres to MVC separation.		
Alternatives	1. Allow the getter methods of the Immutable Interface to have parameters in all situations. 2. Allow the getter methods of the Immutable Interface to have parameters in the situation where the parameter data is not used to trigger a state change in the Business Model, nor used to update the data of the Business Object. **3. Do not allow the getter methods of the Immutable Interface to have parameters.**		
Decision	Option 1 is not allowed, because it violates MVC. Option 1 would allow the user interface to change the state of the Business Model without going through the business controller layer (Business Mediator). Option 2 was not selected, because it introduces too many "what if" scenarios, therefore, complicating the decision tree of what is and isn't acceptable for the Immutable Interface. Goal is to define an prescriptive architecture that is as user		

	friendly as possilbe, and allowing Option 2 would not help us meet this objective. **Option 3 was selected**, because it ensures MVC adherence and does not introduce complex decision trees into the usability of the architecture. Rather than allow the Immutable Interface getter methods to take parameters, the parameters should be passed to Business Mediator Functions, via the Business Contracts.

Mutable Interface

Architectural Decisions:

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Mutable Interface may provide access to public setter methods, service methods that change state, and getter methods that the designer did not want to expose to the access channel but must expose to the Business Mediator and other Business Objects.	BM5.0	Model
Problem Statement	What methods can be exposed on the Mutable Interface?		
Assumptions	• There may be situations where a getter method is inappropriate to expose to the Access Channel, but is necessary to expose to the Business Mediator. • Only public services may be exposed through the Mutable Interface.		
Motivation	Define boundaries of the functions that a Mutable Interface may expose to its users.		
Alternatives	1. Public setter methods. 2. Public getter methods that the user didn't want to expose to the User Interface through the Immutable Interface. 3. Public service methods that change the state of the Business Objects other than the setter methods. 4. Public getter methods that were exposed through the Immutable Interface.		
Decision	Option 1, 2, and 3 are all approved services that may be defined on the Mutable Interface. Option 4 is not allowed. The Mutable Interface will inherit all of the getter methods defined on the Immutable Interface and should not redefine them on itself.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Mutable Interface methods can return native types, simple object types (e.g., String), Mutable Interfaces, Immutable Interfaces, or void.	BM6.0	Model
Problem Statement	What information can the Mutable Interfaces return to their users?		
Assumptions	• Users of the Mutable Interface can have access to objects that provide services that change the state of the Business Model. • Users of the Mutable Interface cannot have access to the Business Objects themselves in order to reduce coupling of objects.		
Motivation	Ensure that the users of the Mutable Interface do not get exposed to the Business Objects themselves through the return value, thus preserving the separation of concern among the layers and objects.		

Alternatives	**1. Mutable Interfaces can provide methods that only return native types, simple object types (e.g., String), Mutable Interfaces, Immutable Interfaces and void.** 2. Mutable Interface can provide setter methods that return all types in option 1 and Business Objects.
Decision	**Option 1 was selected**, because it provides the more strict layering approach and decouples the users of the Mutable Interface from a specific concrete implementation of the model, thus reducing the impact of change. Option 2 was not selected, because it violates the assumption that users of the Mutable Interface cannot have access to the Business Objects. Option 2 would not guarantee us the potential to use the Business Mediator on top of another Business Model.

Business Object

Architectural Decisions:

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Aggregate Business Objects hold reference to the Mutable Interface of parts	BM7.0	Model
Problem Statement	Object oriented solutions are founded on the concept of proper use of aggregation. However, even appropriate use of aggregation between business objects increases the coupling in the system and therefore impacts its' flexibility and maintainability. How can we architect a solution that supports aggregation, but provides flexibility in its implementation?		
Assumptions	• Aggregate whole, Business Objects have a need to modify the state of their parts in specific situations.		
Motivation	Reducing the coupling between the specific implementations of the Business Objects within a business model will reduce the impact of change amongst aggregate whole Business Objects when their parts change. The result, a net improvement in the flexibility and maintainability of the solutions built on R4SC.		
Alternatives	1. Aggregate whole business objects hold onto their parts' Business Object reference. **2. Aggregate whole business objects hold onto their parts' Mutable Interface reference.** 3. Aggregate whole business objects hold onto their parts' Immutable Interface reference, and have the option at any time to cast that reference to its Mutable Interface temporarily within the execution of a method.		
Decision	Option 1 was not chosen because it increases the coupling between the concrete implementation of the other Business Objects in the business model. This makes the solutions less flexible and more susceptible to the impacts of change. **Option 2 was selected**, because it is appropriate for an aggregate whole business object to affect state changes in its aggregate parts. Option 2 is superior to option 1, because it does not result in the direct coupling of an aggregate whole Business Object to the concrete implementation of its' Business Object parts. Rather, it couples the aggregate whole Business Object to a more flexible, abstract definition of its parts – the Mutable Interface. Since the Mutable Interface only defines the method signatures that are available, while allowing for the specific implementation of the interface to vary, this approach is far more		

	flexible than option 1. Additionally, since the Mutable Interface isolates the user from how the Business Object specifically implements its' messages, changes to the Business Object can be better isolated, thus reducing the impact of change to the aggregate whole Business Object. Finally, Option 2's use of Interfaces provides the potential to leverage dependency injection to bind the implementation at run time, rather than design/implementation time. Option 3 was not selected, because while it provides additional decoupling by further limiting the visibility provided by referencing the Mutable Interface, it provides no added benefit to decouple the aggregate whole Business Object from its aggregate parts since option 2 already ensures that the reference is to an abstract interface. Option 3 introduces additional overhead requiring the Business Object to cast its parts' references to Mutable Interfaces in order to make changes in the state of those parts. This overhead is not justified by additional benefits over option 2.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	BusinessObjects must know how to instantiate, retrieve, populate, and persist themselves	BM8.0	Model
Problem Statement	Who should have responsibility for instantiating and populating a Business Object? A common implementation weakness in the development of object oriented solutions is in the external management of an objects lifecycle. We have repeatedly observed solutions in which one object (Object A) performs the task of instantiating another objects (Object B) instances, retrieving that instances (Object B's) information from persistent storage, and decomposing the information from persistence into the other object's (Object B) attributes through the use of Object B's setters. The result is the increase in the coupling of Object A to other, inappropriate objects in the system. In terms of responsibility, Object A becomes too strong. In that same vein, Object B becomes weak, because it doesn't know how to retrieve, populate, and persist itself; therefore, it is not a true Business Object.		
Assumptions	• Objects need to know how to instantiate, retrieve, populate, and persist themselves. • Objects can accomplish these tasks through the use of helper classes, but they are responsible for managing the process and isolating other, non-helper objects from the knowledge of how this is done. • As coupling increases defects and cost of maintenance increase, and flexibility of the solution decreases. All negative affects on the solution. • Solutions that don't adhere to assumption 1 have duplicate business logic and unnecessary increased lines of code.		
Motivation	Development of solutions that are more flexible to change and, subsequently, have a lower cost of ownership.		
Alternatives	**1. Business Objects define public service methods on themselves that others can call to retrieve, populate, and persist.** 2. Aggregate whole, Business Objects are responsible for retrieving, populating, and persisting their part Business Objects. This knowledge is within the aggregate whole. The aggregate part is not aware of how to perform these tasks.		
Decision	Option 1 is the only option.		

Subject Area	**R4SC**	**AD ID#**	**App Area**

Architectural Decision	Each BusinessObject must implement it's corresponding Mutable Interface	BM9.0	Model
Problem Statement	If a Business Object is not going to expose any setter functionality, why not have the Business Object implement the Immutable Interface and not create a Mutable Interface for that Business Object?		
Assumptions	• Consistent design and implementation simplifies a solution. • Over the lifecycle of any solution it is difficult to predict whether an object will need to expose setter functionality, at some time. • Empty Mutable Interfaces add neither implementation nor maintenance overhead; particularly, given MDA transforms are available that will generate these components automatically.		
Motivation	Simplify the solution through consistent implementation.		
Alternatives	1. Business Object implements its Immutable Interface in situations where no setter functionality needs to be exposed on a Mutable Interface. **2. Business Object implements its Mutable Interface.**		
Decision	Option 1 was not chosen because it introduces inconsistency into the design and implementation. Since a majority of the Business Objects do expose setter functionality at some point in the application, other Business Objects are implementing their Mutable Interfaces and we should consistently adhere to that rule, even if the Mutable Interface is empty. As the solution matures, there is a good chance that the blank Mutable Interfaces will be populated to support future business capabilities. **Option 2 was selected**.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Business Objects should have attributes on themselves to hold their data and maintain their state internally.	BM10.0	Model
Problem Statement	How should the Business Object manage its state?		
Assumptions	• The Business Object should manage its data consistently regardless of whether the instance is retrieved from persistence or newly created. • The Business Object cannot have any knowledge of the integrated data sources, middleware, SOA services or operational systems of the enterprise.		
Motivation	Design an Business Object data/state management approach that is consistent, preserves the decoupling of the business logic from the integrated technologies, and upon meeting those criteria is the most efficient to develop/maintain.		
Alternatives	1. Define an attribute on the Business Object to hold the Immutable Data Object Interface reference and method chain the Business Object's getters/setters to the Immutable Data Object Interface getters/setters (best used when the Business Object and Immutable Data Object Interface have similar structures). 2. **Hold all the Business Objects data as attributes on the Business Object**. When retrieving objects from persistent storage, transfer the data from the Immutable Data Object Interface to attributes of the Business Object and release reference to the Immutable Data Object Interface. This can best be accomplished by passing the Immutable Data Object Interface as a parameter to the constructor method of the Business Object, and allowing the constructor to transfer the data within its execution.		
Decision	Option 1 was not selected primarily, because it doesn't meet the consistency criteria. Option 1 only applies for objects that are going to be persisted. Therefore, transient business objects would not have a reason to apply this state		

	management approach. Option 1, through the use of the Immutable Data Object Interface, will ensure that the Business Objects remain independent of the enterprises back-end systems. Option 1 is the more efficient approach from a development and maintenance perspective, but because it cannot be applied consistently for all Business Objects, it was not chosen. Option 2 was selected because it can be applied consistently for both persistent and transient Business Objects. Option 2, like Option 1, also will ensure that the Business Objects remain independent of the intgrated technologies by communicating with the Immutable Data Object Interface. Option 2 does have a little more overhead that Option 1 in the development/maintenance lines of code, as this option requires the constructor on the Business Object transfer the data from the Immutable Data Object Interface, returned by the Integration Mediator Function, to the appropriate attributes on the Business Object instantiated. However, Option 2 provides the advantage of allowing the Data Object to be released thus supporting efficient memory management and garbage collection.

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Instance creation and management of the Business Object must be managed through a factory method.	BM11.0	Model
Problem Statement	Provide guidance for the instantiation and instance management of Business Objects to best meet the needs of a solution throughout its lifespan.		
Assumptions	• Tailoring of instantiation strategies is necessary to address hot spots during performance testing. • It is difficult to anticipate your hot spots proactively, so a good architecture provides flexibility to make changes to address the hot spots reactively. • It is essential to be able to respond to these changes rapidly, as stress/performance testing is often in the critical path to delivery.		
Motivation	Accelerated time to market through flexibility. A solution that delivers cost savings and a positive ROI over the life of the project. We need to make sure that the solution is not more expensive to implement than the change impact that would be expienced with a less encapsulated solution. Over the life of a solution it often becomes necessary to switch instantiation strategies to meet the changing non-functional demands of a solution. The architecture should encapsulate from the requestors the knowledge of whether an instance-based, pooling, or singleton instantiation strategy is being used.		
Alternatives	1. Where an instance-based strategy is deemed appropriate, allow the Business Object to expose the default instantiator as a public method to request instances. 2. Where a singleton strategy is deemed appropriate, allow the Business Object to expose a singleton method. **3. Instance creation and management of the Business Object must be managed through a factory method.**		
Decision	Option 1 locks the Business Object into an instance-based approach. If, at some point in the future, it becomes necessary to transition to a singleton or pooled instantion strategy, the requestors of the Business Object will need to change. In Java, this approach would have the requestor call the new() method on the Business Object.		

	Option 2 locks the Business Object into a singleton strategy, resulting in the same change impact as option 1 if a shift to an instance-based or pooling strategy is required. In this approach, a singleton operation (e.g., getSingleton) would be defined as a class operation on the Business Object. The default instantiator would be made private to prevent anyone from creating new instances. A private attribute (singleton) would be defined on the Business Object to hold reference to the singleton instance. The getSingleton operation would first check to see if the singleton attribute was populated. If it was, it would return reference to that instance. If not populated, it would call the private default instantiator to populate the singleton attribute and return the reference to the instance to the caller. In option 3, you would define a factory method (e.g., getInstance), which would encapsulate knowledge of how the instances are being managed. This approach would require the the default instantiator (e.g., new() in Java) be defined as a private method. If an instance-based strategy was needed for the Business Object, the getInstance method would simply turn around and call the default instantiator returning the instance that was created. For a singleton situation, the getInstance method would check an attribute (e.g., singleton) to see if it was populated. If so, it would return the instance, if not it would call the default instantiator to create an instance, populate the singleton attribute with this instance, and return the reference to the singleton instance to the requestor. Finally, if a pooling scenario is required, the getInstance method would check an attribute that holds the collection of instances (e.g., instancePool) and "check out" one of the instances to return to the requestor. If all the instances were in use the getInstance method could, if allowed, request a new instance. The getInstance method would be responsible for managing the pool size as defined by the architectural decisions for the project. With option 3 the knowledge of providing an instance is encapsulated within the getInstance method, thus allowing the architect to shift from instance to singleton to pooled instantiation strategy without it resulting in a change for requestor, thus addressing the first motivation. The second issue was whether this approach would be cost prohibitive, which this approach is not. The cost of designing and implementing this approach versus option 1 or 2 is nominal in comparison. For that reason, option 3 is required.

Integration Adapter Layer

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Only the Integration Layer of the application can interface with the enterprise data sources.	IL 1.0	Integration Layer
Problem Statement	What layer(s) of the application architecture should be able to interface with the enterprise data sources (e.g., database, legacy applications, B2B apps, etc.)?		
Assumptions	• Reducing the number of touch points to the enterprise data sources improves maintainability. Optimum solution is 1 interface point to that data. • Data source interface management must be separated from the business logic processing.		
Motivation	Reduce the impact of change to a solution built on the architecture, by reducing the points of the architecture that have knowledge of the enterprise data sources.		

	Simplify the architecture by defining a consistent, single approach for accessing the enterprise data source.
Alternatives	1. Allow the user interface to access the enterprise data layer (common implementation approach with JSP/Servlets). 2. Allow the Business Mediator Function to access the enterprise data layer to initially retrieve business objects and allow the Business Objects to interact with the enterprise data layer after the initial retrieval (EJB approach) 3. Allow the Business Objects to access the enterprise data layer. **4. Define an adapter layer (Integration Mediator) that isolates the business logic from the enterprise data layer. Have the Business Objects interact with that adapter layer.**
Decision	Option 1 is the worst alternative. It violates Model-View-Controller and typically results in a single layer of god objects (objects that are extremely coupled to the entire solution space and very brittle to change). Option 2 was not selected as it complicates the architecture. In this approach two layers of the architecture (the business controller and the business model) know how to access the persistent data. With two layers accessing the persistent data, there is also twice as much potential impact when changes occur in the enterprise data layer. Option 3 was not selected, because the business object becomes aware of the specific enterprise data sources it is interacting with, the number of sources, and how to manage units of work across the sources. A better solution would be to isolate this away from the business objects in an adapter, which is the solution in option 4. **Option 4 was selected**, because it results in a separate component for managing the persistent data source interaction and integration that is isolated from any particular business domain. By isolating this functionality from the business model, another application team could leverage the persistence capabilities with their business model. The adapter also helps to reduce the potential for change in the business domain in the event that a change is necessary in the enterprise data layer.

Integration Contract

Architectural Decisions:

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Implementation of Integration Contract can hold references to Immutable Interfaces, but cannot provide an accessor that returns the Immutable Interface	IL2.0	Integration Layer
Problem Statement	How do I provide access to the Business Model information that the Integration Adapter layer needs without coupling the components of the Integration Layer to the business model?		
Assumptions	• Decomposing complex aggregate objects into their simple attributes is complicated when one or more parts are collections. • Coupling the integration layer of an architecture with the business model layer will make the architecture more susceptible to the impacts of change and less flexible.		

	• The Integration Layer should not change the state of a business object (no setter access).
Motivation	Efficiently deliver Business Object information to the Integration Adapter without coupling the Integration Layer to the business model.
Alternatives	1. Decompose the aggregate Business Objects into their simple attributes and populate the Integration Contract implementation with that information. The Integration Contract provides accessor methods that can return native types and simple objects (e.g., String) only. 2. Pass to the Integration Contract implementation the Business Object reference, and don't allow the contract to pass the reference to any other components in the Integration Adapter. The Integration Contract provides accessor methods that can return native types and simple objects (e.g., String) only. 3. Pass to the Integration Contract implementation the Mutable Interface reference, and don't allow the contract to pass the reference to any other components in the Integration Adapter. The Integration Contract provides accessor methods that can return native types and simple objects (e.g., String) only. **4. Pass to the Integration Contract implementation the Immutable Interface reference, and don't allow the contract to pass the reference to any other components of the Integration Adapter. The Integration Contract provides accessor methods that can return native types and simple objects (e.g., String) only.**
Decision	Option 1 is inefficient from a design, implementation, and maintenance standpoint. Option 2 couples the Integration Contract implementation to a concrete Business Object, providing for no flexibility in the implementation. Additionally, Option 2 exposes to the Integration Contract all public setter functions, thus allowing the Integration Contract to change the state of the Business Object. Option 3 provides the Integration Contract implementation access to change the state of the Business Object through the Mutable Interface. **Option 4 was selected**, because it provides flexibility of the implementation (as the Immutable Interface is an abstract implementation of the Interface pattern) and it does not expose to the Integration Contract the functions to change the state of the Business Model.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Contracts are only allowed to contain getter methods and inquiry methods (e.g., "is…")	IL3.0	Integration Layer
Problem Statement	How can we provide a component that encapsulates the information inquiry needs of the Integration Mediator Function in order to allow it to perform its job?		
Assumptions	• Integration Contract is input to an Integration Mediator Function for its use. • Integration Contract content should not be mutable by the Integration Mediator Function.		
Motivation	The Integration Contract is written from the perspective of the Integration Mediator Function's input requirements. It should not be used as a vehicle to		

	pass information back to the requestor. To eliminate this possible misuse, the Integration Contract will not be allowed to define any setter functionality.
Alternatives	1. Allow the Integration Contract to contain setter methods, thus allowing the Integration Mediator Function to add information gathered during its execution. **2. Allow the Integration Contract to provide only getter methods, thus reducing the contract to a unidirection vessel for passing information to the Integration Mediator Function for its consumption.**
Decision	Option 1 was not selected, because it complicates the usability of the contract. This approach results in contracts with getter and setter methods defined for two different users: the Integration Mediator Function and the requestor. What information must the requestor provide to the Integration Mediator Function for it to execute successfully? Which of the getters is provided for the requestor to receive information? When can the requestor expect to be able to call the getter methods whose data is populated by the Integration Mediator Function and have valid information? **Option 2 was selected**, because it provides the most user friendly alternative. The contract has only one user: the Integration Mediator Function. All the getter methods on the Integration Contract are expected by the Integration Mediator Function, and therefore, must be provided by the requestor (no question as to who provides information and when it needs to be available). Forces the Integration Mediator Function to use the return value to provide information back to the requestor, which is a better practice than placing return information on the contract.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Contracts should be designed for use by a single Integration Mediator Function.	IL4.0	Integration Layer
Problem Statement	Should we create more generic, broader contracts that can be leveraged by several Integration Mediator Functions, or should we be specific about the data needs for a operation and limit the usability of the contract to a single Integration Mediator Function?		
Assumptions	• Optional information increases the complexity of the contract. Makes it more complicated to document and explain for use by end users. • Optional information increases the likelihood of implementation errors, thus increasing the complexity of the contract validation that needs to be performed. • Case statements to support the business rules of optional information are very susceptible to change, and in many situations the case statements for different usage scenarios will diverge over time, making the logic more complicated and more difficult to maintain. • Optional information cannot be eliminated for all cases. • Very specific contracts that limit or eliminate the optional data reduce the potential for their reuse, thus increasing the number of classes that need to be maintained in the solution. • Unique contracts per Integration Mediator Function eliminate the possibility of a shared contract having to be split into two in the future when the information rules for the various Integration Mediator Functions using the contract diverge. By eliminating this possibility, the service signatures will		

	be more stable over time, thus improving the maintainability of the business objects that leverage the Integration Mediator Function.
Motivation	Simplify the usability of the Integration Mediator Functions for the users that will have to design to their API. Provide a solution that provides the most stable Integration Mediator Function signature for the service users to minimize the impact of change to the user throughout the life of the solution.
Alternatives	1. Leverage a single Integration Contract that contains accessor methods to all the information that will be needed by any Integration Mediator Function in the system. 2. Share Integration Contracts across Integration Mediator Functions that have similar purposes to reduce the number of contracts that need to be maintained and take advantage of reuse. **3. Define a Integration Contract for each Integration Mediator Function.**
Decision	Option 1 was not selected for several reasons: the contract implementation would be excessively large, which would be disadvantageous in a distributed environment. The contract would be complicated for users to design to, because the specific data needs for the service they need to leverage aren't obvious in the definition of the class. The documentation required to describe the data needs per Integration Mediator Function would be difficult to develop, cumbersome to read through for the user, and hard to maintain as service needs change and as services are added. Option 2 was not selected due to the risk that over time the information needs of even similar Integration Mediator Functions could change; thus, requiring the contracts to be broken up, which would cause the service method signatures to change. The potential negative impact of this change effect to the various users of the Integration Mediator Functions are greater than the time saved by reducing the number of contracts in the system through reuse of the contracts. **Option 3 was selected** because it provides the most straight forward definition of the information needs of an Integration Mediator Function to the users of that service. Since the contract is used by only one Integration Mediator Function, optional data will be significantly reduced. Option 3 also eliminates the possibility that over time an Integration Mediator Function's method signature will change due to it's contract type changing. Since only one Integration Mediator Function uses a particular contract, the contract content can change to support the changing information needs of that Integration Mediator Function. There is no risk that those information needs will diverge from another Integration Mediator Function's needs, since no other Integration Mediator Function leverages the contract.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Contracts encapsulate the information needs of the Integration Mediator Function and are abstract implementations of the Interface pattern.	IL5.0	Integration Layer
Problem Statement	How can the requests for information required by the Integration Mediator Function to perform its operation be defined, while providing maximum flexibility in the implementation for the requestors?		
Assumptions	• Integration Mediator Functions will be independent of the knowledge of those requesting its service.		

	• Implementation flexibility is critical to allow users to take advantage of various application distribution architectures and technologies.
Motivation	Design an Integration Mediator Function that can be leveraged across requestors; thus, 1) ensuring consistency of response across various requestors and 2) reducing maintenance by eliminating redundant silos of integration logic.
Alternatives	1. Create a concrete class that encapsulates the requests for information and place an instance of this class as the parameter for the Integration Mediator Function. **2. Leverage the interface pattern to create an abstract definition of the requests for information that the Integration Mediator Function expects answered. Place an instance of this interface as the parameter for the Integration Mediator Function.**
Decision	Option 2 was selected, because it allows for implementation flexibility, thus better addressing the second assumption.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Contract methods can return native types and simple object types (e.g., String) only	IL6.0	Integration Layer
Problem Statement	We must ensure that we do not create a presumed dependency for order of execution of requestors accessing the Integration Adapter layer.		
Assumptions	• Cannot define the Integration Contract methods to require a return type that would only be accessible by executing another request to the Integration Layer first.		
Motivation	Eliminating an order of operation prerequisite for the Integration Mediator Functions is essential to delivering a flexible SOA architecture.		
Alternatives	**1. Allow the Integration Contract methods to return native types and simple object types only.** 2. Allow the Integration Contract methods to return native types and simple object types (Alternative 1), and also allow them to return Immutable Data Object Interfaces.		
Decision	Option 1 was selected, because the only way for a instantiator of a Integration Contract to have access to an Immutable Data Object Interface would be to require another request to the Integration Mediators be executed first. Such a dependency on the order of operation should be encapsulated within the Integration Mediator layer to eliminate potential misuse and simplify use of the Integration Mediators for the users.		

Integration Mediator

<u>Architectural Decisions:</u>

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	All logical units of work to manage the integration of disparate back-end technology and any physical transactions must be started, executed, and closed within the execution of the Integration Mediator Functions	IL7.0	Integration Layer
Problem Statement	Where should transaction management occur within the layered architecture?		
Assumptions	• Knowledge of the underlying enterprise and database products/technologies		

	cannot pass beyond the Integration Adapter Layer.
Motivation	Ensure that the Integration Mediator Functions are independent and may be executed without concern of order.
Alternatives	**1. Allow logical transactions to span the breadth of many Business Mediator Function and allow physical transactions to span the run of a single Integration Mediator Function.** 2. Allow logical transaction to run for the duration of a single Business Mediator Function request and allow physical transactions to run for the duration of a single Integration Mediator Function. Full lifecycle of the transaction must process within the execution of a single Business Mediator Function.
Decision	**Option 1 was selected**: Transactional capabilities should not be limited to specific methods or technology layers in order to provide an acceptable level of flexibility and independence for the framework. The only limitation is that the logical transaction management cannot extend above the Business Mediator Function. Physical transactions against the data sources, likewise, should not be allowed to extend above the Integration Mediator Function.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Mediator Functions must take as their only parameter an Integration Contract	IL8.0	Integration Layer
Problem Statement	Provide a simple, consistent signature for communicating with the Integration Mediator Function that will reduce the impact of its changing information needs.		
Assumptions	• Simplifying the interface between the business domain and the integration logic improves the communication between these teams, which has a positive affect on integration quality and the possibility of parallel development.		
Motivation	Reduce maintenance costs by simplifying the architecture and decoupling the layers making them less susceptible to the impacts of change.		
Alternatives	**1. Pass all information to the Integration Mediator Function in an object (Integration Contract) that encapsulates all its' information needs. The specific parameter type expected should be an abstract implementation of the Interface pattern to allow for flexibility in the implementation.** 2. Expose each piece of information needed by the Integration Mediator Function as a separate parameter on the method signature.		
Decision	**Option 1 was selected**. Making sure that the Integration Contract is an abstract implementation of the Interface pattern, rather than a specific, concrete class type is critical to allow the requestors flexibility in how they implement the contract. Option 2 was not selected because it exposes all the information that the Business Mediator Function will use in the method signature. When the information needs of the Business Mediator Function change, the requestor will need to modify the operation that it calls to the updated signature. Implementation is complicated, as well, as the developer must confirm that the order of the attributes passed is consistent with the order specified in the signature.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Mediator Functions can return one of five values: void, an Immutable Data Object Interface, a collection of Immutable Data Object Interfaces, a	IL9.0	Integration Layer

	native type, a collection of native types		
Problem Statement	What is the best way for the Integration Mediator Function to respond to the requests of the Business Model layer with respect to separation of concern and flexibility to future requirements changes?		
Assumptions	• The business domain should know as little as possible, internally, about the integration specifics of the solution. • Proper layered architecture best practices prohibit requests from a lower layer of the architecture directly to a higher layer in the architecture (Integration Layer components cannot make requests to the Business Domain). • Information needs to be returned to the requestor in the business domain from the requested Integration Mediator Function.		
Motivation	Ensure consistency of return information from the Integration Mediator. Preserve separation of concern between the Business Model and Integration Adapter.		
Alternatives	1. Allow the Integration Mediator Function to place the data retrieved during its processing directly onto the Business Object, either by directly having access to the Business Object or through the use of the Mutable Interface. Requires that the reference to the Business Object or Mutable Interface be passed into the Integration Mediator Function on the Integration Contract. In the case of an initial retrieve being performed, the Integration Mediator would know how to get a new instance of the Business Object. No return value for the Integration Mediator service methods required. 2. Have the Integration Mediator Function return native types, simple Objects (e.g., String), or Data Objects to the requesting Business Object. Have the Business Object internally know how to process this information. **3. Have the Integration Mediator Function return native types, simple objects (e.g., String) or Immutable Data Object Interfaces to the Business Object. Have the Business Obejct internally know how to process the Immutable Data Object Interface information.**		
Decision	Option 1 is the poorest solution. It requires that the Integration Layer (a lower layer in the architecture) trigger requests to the Business Model layer (a layer higher in the architecture), which breaks best practices of layered architecture that a lower layer should not start conversation with a higher layer. This solution also tightly couples the Integration Mediator Function with the Business Object/Mutable Interface, by taking the responsibility of populating he Business Object away from the Business Object, itself, and placing it in the Integration Mediator (a violation of architectural decision *BM8.0, Business Objects must know how to instantiate, retrieve, populate, and persist themselves*., above Option 2 was not selected, because the Data Object is a concrete class that provides zero flexibility in the implementation. Having the Business Object deal directly with a concrete class rather than an abstract implementation of the Interface pattern limits the Business Object to only be deployed onto of the current Integration Layer, something that should be allowed to change as the needs of the business evolve. This solution will not allow for the Integration Layer to change without significant change impact in the Business Model. **Option 3 was selected**, because it guarantees separation of concern and adheres to architecture layering best practices. Additionally, it provides greater flexibility in the implementation than option 2, as it returns an abstract implementation of the Interface pattern (Immutable Data Object Interface) rather		

	than a concrete Data Object.

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	Integration Mediator Functions can pass reference to the Integration Contract to objects within the Integration Layer.	IL10.0	Integration Layer
Problem Statement	Should the Integration Mediator Function be allowed to pass the Integration Contract to the Data Objects, since the Integration Contract represents the input data that the Data Objects require to communicate with the Integration Adapter ?		
Assumptions	• Unlike the Business Mediator Function decision to not allow the Business Contract to be forwarded outside of the Business Controller Layer, this decision relates to passing the Integration Contract within the Integration layer. • Integration Contracts are designed for consumption by the Integration Mediator Function to fulfill that service's information requirements. • Forcing a Data Object to use a particular contract may limit its' reusability among Integration Mediators and across the organization. • Integration Contracts can contain information that an Integration Mediator Fucntion will use across several Data Object requests.		
Motivation	Design an architecture that is flexible to changing business needs.		
Alternatives	**1. Allow the Integration Mediator Function to forward the Integration Contract to the Data Object or helper classes in the Integration Layer. The Data Object or helper classes can then pick the data that they need.** 2. Extract the content from the Integration Contract via the Integration Mediator Function. Allow the Integration Mediator Function to forward the required information to the appropriate Data Object to execute the request. The Integration Mediator Function drops reference to the Integration Contract at the conclusion of its execution. 3. Extract the content from the Integration Contract via the Integration Mediator Function. Have the Integration Mediator Function retrieve an instance of the appropriate Data Object or helper class and then have the Integration Mediator Function place the appropriate information on that object using its setter methods.		
Decision	Option 2 was not selected for two related reasons: 1) option 2 is inefficient from a design, development, and run time perspective when compared to the option of passing the contract, and 2) option 2 would either result in method signatures with a large number of parameters. Methods with a large number of parameters are cumbersome to integrate and as the data needs change for the method the parameter list will change causing a change impact to each requestor of the method. Option 3 was not selected, because it too is inefficient, and by making the Integration Mediator Function instantiate and set the data on the Data Object the Integration Mediator Function becomes unnecessarily coupled to the Data Object. This option also removes the knowledge to populate the Data Object from itself, which is counter to object-oriented best practices. **Option 1 was selected**, because it allows us to keep the parameter list on the Data Objects simple, does not tightly couple the Integration Mediator to the Data Object, and keeps the knowledge of populating the Data Object within the Data Object, itself. Additionally, this solution is the most efficient approach.		

	With Option 1, In the event that multiple Integration Mediator would need to use the same Data Object, there would be a need for that Data Object to provide separate methods that have as a parameter one of the expected contracts (e.g., retrieve(ContractA) and retrieve(ContractB).

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	An Integration Mediator Function can leverage other Integration Mediator Functions to delivery its operation.	IL11.0	Integration Layer
Problem Statement	Should Integration Mediator Functions be self-contained and depend on no external Integration Mediator Functions, or should a Integration Mediator Function reuse other Integration Mediator Functions.		
Assumptions	• Objects in the same layer can have public access to one another without violating the best practices of layered architecture. • Best practices propose the reuse of functions rather than the duplicating the lines of code. • Duplicated lines of code increase complexity and hurt maintainability. • Reusing services from another object couple you to that object. As coupling increases between objects, so does the potential impact of change to the calling object when the called object changes.		
Motivation	Develop Integration Mediator Functions that are easy to maintain and resilient to the impact of change.		
Alternatives	**1. Allow Integration Mediator Functions to leverage Mediator Functions on its Integration Mediator and other Integration Mediators.** 2. Don't allow Integration Mediator Functions to leverage other Integration Mediator Functions, rather have the Integration Mediator Function be self-contained by duplicating the logic within itself. 3. Allow Integration Mediator Functions to leverage other Integration Mediator Functions on its Integration Mediator only.		
Decision	Option 2 was not selected, because it results in duplicating lines of code to perform the same function. This is inefficient from a development and maintenance standpoint, and poor design and implementation. It does provide the maximum decoupling among the Integration Mediator Functions, but the expense is too great. Option 3 was not selected for the same reason as option 2. Option 3 provides a little more flexibility in reusing services, while retaining the same degree of decoupling. But the expense to the development and maintenance are too great. **Option 1 was selected**, because it follows the recommended best practice of reusing operations within the same layer to improve development and maintenance efficiency, and reduce the complexity of the solution by isolating identical functionality to a shared operation. Option 1 does increase the potential of coupling amongst the Integration Mediator Functions, which can have a negative effect on the impacts of change to the solution, however, the impact can be isolated within the Integration Mediator Function itself in most cases. Therefore, the potential of the change impact reaching the users (the Business Objects) of the service is no greater with option 1, than option 2 or 3.		

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	Integration Mediators are not intended to have a single Integration Mediator Function	IL12.0	Integration Layer
Problem Statement	What should a Integration Mediator contain? Should it have only a single Integration Mediator Function?		
Assumptions	• Object metric best practices suggest that throughout a project, objects should have an average of 10-15 operations, with an average of 15 lines of code per operation.		
Motivation	Develop Integration Mediators that logically group Integration Mediator Functions of the application to simplify the usage of the system and correlate with the real-world business process relationships.		
Alternatives	1. An Integration Mediator is defined for each Integration Mediator Function. **2. An Integration Mediator consists of a group of related Integration Mediator Functions.** 3. An Integration Mediator consists of Integration Mediator Functionss for a single Data Object or hierarchy of Data Objects.		
Decision	Option 1 was not selected, because it creates an unnecessarily large number of classes and does not result in groupings of like services. This option is less efficient from an implementation/maintenance standpoint, as it tends to increase the lines of code. Option 3 was not selected either. Integration Mediators should be looked at as groupings of like services, rather than grouping of services for a Data Object. The concern with this approach is that taking this viewpoint may lead to the Integration Mediator doing nothing more than method chaining to the Data Model, which results in integration processing logic transferring to the Data Model. As this process information transfers, the coupling within the Data Model will increase, which will make it more brittle. **Option 2 was selected**, because it results in Integration Mediator Functions that are bundled similar to the way the business views their primary processes being bundled. It results in service-oriented objects that focus on managing the processes and directing the objects to work together to deliver the services. It does not result in an Integration Mediator per Integration Mediator Function, nor a single Integration Mediator for all the Integration Mediator Functions.		

Immutable Data Object Interface

Architectural Decisions:

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	Immutable Data Object Interface will adhere to the guidelines of the Interface pattern and be implemented by it's Data Object	IL13.0	Model
Problem Statement	How can we create a view of the Data Model that can be returned by the Integration Mediator Function to the Business Object, but does not tie the Business Object to a specific implementation of the Data Model?		
Assumptions	• Implementation cannot provide access to methods that would change the state of the Data Model (e.g., setter methods).		

Motivation	Architect a solution that preserves MVC separation between the Business Domain and the Data Domain in the most efficient means possible.
Alternatives	1. Create an implementation of the Proxy pattern to be returned from the Integration Mediator Function to the Business Object. Attributes of the Data Object will be transferred to the lightweight, Proxy instance. **2. Create an implementation of the Interface pattern to be implemented by the Data Object. Allow the Interface to be returned by the Integration Mediator Function to the Business Object.**
Decision	Option 1 provides a lightweight picture of the Data Object that can be beneficial when deployed on a distributed environment. Option 1 also increase the number of classes that need to be developed/maintained, and the number of instances that need to be instantiated and managed during run time. Typically, there is not a lot of extraneous data that is on the Data Object that would not be available for the Business Object, so the Proxy itself is not much more lightweight than the original Data Object. Therefore, the cost of maintaining the additional classes and the performance degradation related to instantiating, populating, and garbage collecting the Proxy are not worth the lightweight benefits that might be delivered for a distributed application. **Option 2 was selected**. It provides better flexibility as the returned value is an Interface (an abstract definition of message signatures), rather than a concrete Data Object or Proxy. The interface actually represents a different view on the Data Object, which means no additional objects need to be instantiated and populated, thus improving performance. While Option 1 was not selected as the preferred approach, there may be situations where it is superior to Option 2, such as distributed architecture situations where the payload sent across the network can be significantly reduced via a Proxy. Through performance modeling, architects should assess their particular situation to determine which alternative is best for their situation.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Data Object Interfaces are only allowed to contain getter methods and inquiry methods (“is…”).	IL14.0	Integration Layer
Problem Statement	How can we create a view of the Data Model that can be returned by the Integration Mediator Function to the Business Objects, but does not tie the Business Objects to a specific implementation of the Data Model?		
Assumptions	• Implementation cannot provide access to methods that would change the state of the Data Model (e.g., setter methods). • Information returned to the Business Objects regarding the Data Model should not include access to integration technology specific information (e.g., relational database, SOA service, operational system ties, etc.).		
Motivation	Enable the Business Objects to be leveraged across different data models that leverage different integration technologies, without the need to change the Business Objects themselves to accommodate the various data models.		
Alternatives	1. Create an implementation of the Proxy pattern to be returned from the Integration Mediator Function to the Business Objects. Data about the Data Object will be transferred to the lightweight, Proxy instance. **2. Create an implementation of the Interface pattern to be**		

	implemented by the Data Object. Allow the Interface to be returned by the Integration Mediator Function to the Business Objects.
Decision	Option 1 provides a lightweight picture of the Data Object that can be beneficial when deployed on a distributed environment. Option 1 also increase the number of classes that need to be developed/maintained, and the number of instances that need to be instantiated and managed during run time. Typically, there is not a lot of extraneous data that is on the Data Object that would not be available for the Provider Router, so the Proxy itself is not much more lightweight than the original Data Object. Therefore, the cost of maintaining the additional classes and the performance degradation related to instantiating, populating, and garbage collecting the Proxy are not worth the lightweight benefits that might be delivered for a distributed application. **Option 2 was selected**. It provides better flexibility as the returned value is an Interface (an abstract definition of message signatures), rather than a concrete Data Object or Proxy. The interface actually represents a different view on the Data Object, which means no additional objects need to be instantiated and populated, thus improving performance.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Data Object Interface getter methods can return native types, simple object types (e.g., String), or other Immutable Data Object Interfaces	IL15.0	Integration Layer
Problem Statement	How can we ensure that the business objects don't become coupled to one implementation of the Integration Model?		
Assumptions	• Requestors cannot have access to any object that can modify the state of the Data Object (e.g, the Data Object, itself). • Requestors cannot have access to any object that can provide it access to an object that can modify the state of the Data Model (e.g, objects that can return Data Objects). • Requestors receive information about the Data Model layer via the Immutable Data Object Interface.		
Motivation	Business models should be reusable on top of different Integration Mediators without having to change the Business model logic.		
Alternatives	**1. Immutable Data Object Interfaces can provide getter methods that only return native types, simple object types (e.g., String), and other Immutable Data Object Interfaces.** 2. Immutable Data Object Interface can provide getter methods that return all types in option 1 and other Data Objects.		
Decision	**Option 1 was selected**, because it ensures that the Business Object will never gain access to the specific implementations of the Data Objects, but rather will integrate with the abstract Immutable Data Object Interfaces (which are implementations of the Interface Pattern). This provides greater flexibility to leverage the Business Model across various Integration Mediators as business needs dictate. Option 2 was not selected, because it allows the Business Model to gain access to the concrete implementation of the Data Model through the Immutable Data		

	Object Interface. This tightly couples the Business Model to that Data Model, therefore, limiting the reusability of the Business Model across Integration Mediators. Option 2 also complicates the architecture, as this option allows both the Business Object and the Integration Mediator to interface with the Data Objects and apply changes, which introduces the questions which one should I use and when.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Data Object Interfaces getter methods should not have any parameters.	IL16.0	Integration Layer
Problem Statement	Should Immutable Data Object Interface getter methods be allowed to take parameter input data?		
Assumptions	• Parameter input data is typically used to trigger a state change or update the internal data of the Data Model. • The Business Model should not be allowed to make state or data changes to the Data Model without going through the Integration Mediator Functions. • Ensuring the separation of the Business Model from the Data Model is crucial for the maintainability and flexibility of the solution.		
Motivation	Design an architecture that ensures separation between the Business Model and the Data Model.		
Alternatives	1. Allow the getter methods of the Immutable Data Object Interface to have parameters in all situations. 2. Allow the getter methods of the Immutable Data Object Interface to have parameters in the situation where the parameter data is not used to trigger a state change in the Data Model, nor used to update the data of the Data Object. **3. Do not allow the getter methods of the Immutable Data Object Interface to have parameters.**		
Decision	Option 1 is not allowed, because it violates Business Model and Data Model separation. Option 1 would allow the business model to change the state of the data model without going through the Integration Mediator Function. Option 2 was not selected, because it introduces too many “what if” scenarios, therefore, complicating the decision tree of what is and isn’t acceptable for the Immutable Data Object Interface. Goal is to define an architecture that is as user friendly as possilbe, and allowing Option 2 would not help us meet this objective. **Option 3 was selected**, because it ensures separation between the business model and the data model and does not introduce complex decision trees into the usability of the architecture.		

Data Object

Architectural Decisions:

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Aggregate Data Objects hold reference to Immutable Data Object Interface of parts (can cast reference to it’s Data Object temporarily within the execution of it’s methods, when	IL17.0	Integration Layer

	needed.)		
Problem Statement	Service oriented solutions are founded on the concept of proper use of aggregation. However, even appropriate use of aggregation between Data Objects increases the coupling in the system and therefore impacts its' flexibility and maintainability. How can we architect a solution that supports aggregation, but provides flexibility in its implementation?		
Assumptions	• Aggregate whole, Data Objects have a need to modify the state of their parts in specific situations. • Of the solutions developed using R4SC, we have observed that the need to access the Data Object is limited.		
Motivation	Reducing the coupling between the specific implementations of the Data Objects within an integration model will reduce the impact of change amongst aggregate whole Data Objects when their parts change. Net improvement in the flexibility and maintainability of the solutions built on R4SC.		
Alternatives	1. Aggregate whole Data Objects hold onto their parts' Data Object reference. **2. Aggregate whole Data Objects hold onto their parts' Immutable Data Object Interface reference, and have the option at any time to cast that reference to its Data Object temporarily within the execution of a method.**		
Decision	Option 1 was not chosen because it increases the coupling between the concrete implementation of the other Data Objects in the data model. This makes the solutions less flexible and more susceptible to the impacts of change. **Option 2 was selected**, because it does not result in the direct coupling of an aggregate whole Data Object to the concrete implementation of its Data Object parts. Rather, it couples the aggregate whole Data Object to a more flexible, abstract definition of its parts – the Immutable Data Object Interface. Since the Immutable Data Object Interface only defines the method signatures that are available, while allowing for the specific implementation of the interface to vary this approach is far more flexible than option 1. Additionally, since the Immutable Data Object Interface isolates the user from how the Data Object specifically implements its' messages, changes to the Data Object part can be better isolated, thus reducing the impact of change to the aggregate whole Data Object.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Data Objects must know how to instantiate, retrieve, populate, and persist themselves	IL18.0	Integration Layer
Problem Statement	Who should have the responsibility to instantiate and populate a Data Object? A common implementation weakness in the development of solution oriented solutions is in the external management of an objects lifecycle. We have repeatedly observed solutions in which one object (Object A) performs the task of instantiating another objects (Object B) instances, retrieving that instances (Object B's) information from persistent storage, and decomposing the information from persistence into the other object's (Object B) attributes through the use of Object B's setters. The result is the increase in the coupling of Object A to other, inappropriate objects in the system. Object B becomes weak, because it doesn't know how to retrieve, populate, and persist itself; therefore, it is not a true Data Object.		
Assumptions	• Objects need to know how to instantiate, retrieve, populate, and persist		

	themselves. • Objects can accomplish these tasks through the use of helper classes, but they are responsible for managing the process and isolating other, non-helper objects from the knowledge of how this is done. • As coupling increases defects and cost of maintenance increase, and flexibility of the solution decreases. All negative affects on the solution. • Solutions that don't adhere to assumption 1 have duplicate logic and unnecessary increased lines of code.
Motivation	Development of solutions which are more flexible to change and, subsequently, have a lower cost of ownership.
Alternatives	**1. Data Objects define public service methods on themselves that others can call to retrieve, populate, and persist.** 2. Aggregate whole Data Objects are responsible for retrieving, populating, and persisting their part Data Objects. This knowledge is within the aggregate whole. The aggregate part is not aware of how to perform these tasks. A common implementation weakness in the development of solution oriented solutions is in the external management of an objects lifecycle. We have repeatedly observed solutions in which one object (Object A) performs the task of instantiating another objects (Object B) instances, retrieving that instances (Object B's) information from persistent storage, and decomposing the information from persistence into the other object's (Object B) attributes through the use of Object B's setters. The result is the increase in the coupling of Object A to other, inappropriate objects in the system. In effect, Object A becomes too stroing, and Object B becomes too weak, because it doesn't know how to retrieve, populate, and persist itself, therefore it is not a true Data Object.
Decision	Option 2 is not allowed, because it results in tight coupling of aggregate whole Data Objects to its aggregate part Data Objects, and removes essential information that should be internal to the part Data Objects. The increased coupling results in solutions that are brittle to change. Additionally, by removing information that the part Data Object should know internally, we have introduced a situation where multiple users of the aggregate part Data Object will have to duplicate the logic that was externalized from it, thus increasing the lines of code of the solution (e.g., if a Data Object doesn't know how to populate itself and two aggregate whole Data Objects use this Data Object as an aggregate part, both aggregate whole objects will need to implement the logic to populate the part).

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Instance creation and management of the Data Object must be managed through a factory method.	IL19.0	Integration Layer
Problem Statement	Provide guidance for the instantiation and instance management of Data Objects to best meet the needs of a solution throughout its lifespan.		
Assumptions	• Tailoring of instantiation strategies is necessary to address hot spots during performance testing. • It is difficult to anticipate your hot spots proactively, so a good architecture provides flexibility to make changes to address the hot spots reactively. • It is essential to be able to respond to these changes rapidly, as stress/performance testing is often in the critical path to delivery.		
Motivation	Accelerated time to market through flexibility. A solution that delivers cost savings and a positive ROI over the life of the project. We need to make sure		

	that the solution is not more expensive to implement than the change impact that would be expienced with a less encapsulated solution. Over the life of a solution it often becomes necessary to switch instantiation strategies to meet the changing non-functional demands of a solution. The architecture should encapsulate, from the requestors, the knowledge of whether an instance-based, pooling, or singleton instantiation strategy is being used.
Alternatives	1. Where an instance-based strategy is deemed appropriate, allow the Data Object to expose the default instantiator as a public method to request instances. 2. Where a singleton strategy is deemed appropriate, allow the Data Object to expose a singleton method. **3. Instance creation and management of the Data Object must be managed through a factory method.**
Decision	Option 1 locks the Data Object into an instance-based approach. If, at some point in the future, it becomes necessary to transition to a singleton or pooled instantion strategy, the requestors of the Data Object will need to change. In Java, this approach would have the requestor call the new() method on the Data Object. Option 2 locks the Data Object into a singleton strategy, resulting in the same change impact as option 1 if a shift to an instance-based or pooling strategy is required. In this approach, a singleton operation (e.g., getSingleton) would be defined as a class operation on the Data Object. The default instantiator would be made private to prevent anyone from creating new instances. A private attribute (singleton) would be defined on the Data Object to hold reference to the singleton instance. The getSingleton operation would first check to see if the singleton attribute was populated. If it was it would return reference to that instance, if not populated, it would call the private default instantiator to populate the singleton attribute and return the reference to the instance to the caller. In option 3, you would define a factory method (e.g., getInstance), which would encapsulate knowledge of how the instances are being managed. This approach would require the the default instantiator (e.g., new() in Java) would be defined as a private method. If an instance-based strategy was needed for the Data Object, the getInstance method would simply turn around and call the default instantiator returning the instance that was created. For a singleton situation, the getInstance method would check an attribute (e.g., singleton) to see if it was populated. If so, it would return the instance, if not it would call the default instantiator to create an instance, populate the singleton attribute with this instance, and return the reference to the singleton instance to the requestor. Finally, if a pooling scenario is required, the getInstance method would check an attribute that holds the collection of instances (e.g., instancePool) and “check out” one of the instances to return to the requestor. If all the instances were in use the getInstance method could, if allowed, request a new instance. The getInstance method would be responsible for managing the pool size as defined by the architectural decisions for the project. So, option 3 addresses the issue of encapsulation and provides a very flexible approach, thus addressing the first motivation. The second issue was whether this approach would be cost prohibitive, which this approach is not. The cost of designing and implementing this approach versus option 1 or 2 is nominal in

	comparison. For that reason, option 3 is required.

Integration Adapter

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	All physical transactions are must be started, executed, and closed within the execution of the Integration Adapter.	IL20.0	Integration Layer
Problem Statement	Which component should have specific knowledge of the integration technologies physical transaction model?		
Assumptions	• Knowledge of the integrated technologies should be limited to a single component to provide the greatest encapsulation.		
Motivation	Encapsulate the complexity of coordinating integration technologies to retrieve data from the user of the Integration Mediator.		
Alternatives	1. The Integration Mediator Function is responsible for managing the physical transactions. 2. The Data Object is responsible for managing the physical transactions. **3. The Integration Adapter is responsible for managing the physical transactions.**		
Decision	**Option 3 was selected**: Was selected because it is responsible for managing and encapsulating all the specific integration with the integration technologies. Option 1 and 2 were not selected, because this would result in additional components of the architecture becoming coupled to the integrated technologies. In the even that those technologies had to change, the impact of change would be greater as the number of entities coupled to them increase.		

Architectural Decisions: Intregration Development Scenario

Provider Router

Architectural Decisions:

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	The Provider Router will have the responsibility for session management.	AI1.0	Service Controller
Problem Statement	Which layer(s) of the application should have access to session management?		
Assumptions	• Solution should reduce coupling within the architecture to the session management common service.		
Motivation	Decoupling the business logic of the solution from technology/products. Providing flexibility to the users of the business logic.		
Alternatives	1. Isolate the session management to the Business Controller layer so that the session management only needs to be implemented once, consistently for all users of the Business Mediator. 2. **Push the session management to the Provider Router layer allowing for each user interface to manage session as it needs.**		
Decision	Option 1 was the initial approach chosen during early implementations using this technique, however, through experience option 1 has been ruled out. Option 1 was initially selected, because it provided development efficiency by implementing the session management once for all users of the Business Mediators. The drawback to this approach was that it required the business controller team to: 1) introduce technology specifics into the Business Mediator space (e.g., leverage the WAS session management component), or 2) forced the development of a custom session management approach to isolate the Business Mediator from the session management. Introducing technology into the Business Mediator violates a primary objective of architecture – to remain technology independent from the Controller Layer to the Integration Adapter Layer. Developing an internal session management approach was time consuming and introduced risk/maintenance overhead that wouldn't be present when leveraging the many available session management components provided in common software packages (e.g., WebSphere). An additional drawback is that this approach assumes that session management requirements will be consistent across all user interfaces, which has proven not to be a safe assumption. **Option 2 is the selected approach**, because it removes technology coupling from the Business Mediator. Additionally, by placing the session management in the hands of the Provider Router, we allow the user interface layer the flexibility to take advantage of session management services that are optimized for the UI technologies use (e.g., WAS Session Management for JSP/Servlets).		

Integration Contract

Architectural Decisions:

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Contracts are only allowed to contain getter methods and inquiry methods (e.g., "is…")	IL1.0	Integration Layer
Problem Statement	How can we provide a component that encapsulates the information inquiry needs of the Integration Mediator in order to allow it to perform its job?		
Assumptions	• Integration Contract is input to the Integration Mediator for its use. • Integration Contract content should not be mutable by the Integration Mediator.		
Motivation	The Integration Contract is written from the perspective of the Integration Mediator's input requirements. It should not be used as a vehicle to pass information back to the requestor. To eliminate this possible misuse, the Integration Contract will not be allowed to define any setter functionality.		
Alternatives	1. Allow the Integration Contract to contain setter methods, thus allowing the Integration Mediator to add information gathered during its execution. **2. Allow the Integration Contract to provide only getter methods, thus reducing the contract to a unidirection vessel for passing information to the Integration Mediator for the service's consumption.**		
Decision	Option 1 was not selected, because it complicates the usability of the contract. This approach results in contracts with getter and setter methods defined for two different users: the Integration Mediator and the requestor. What information must the requestor provide to the Integration Mediator for it to function successfully? Which of the getters is provided for the requestor to receive information? When can the requestor expect to be able to call the getter methods whose data is populated by the Integration Mediator and have valid information? **Option 2 was selected**, because it provides the most user friendly alternative. The contract has only one user: the Integration Mediator. All the getter methods on the Integration Contract are expected by the Integration Mediator, and therefore, must be provided by the requestor (no question as to who provides information and when it needs to be available). Forces the Integration Mediator to use the return value to provide information back to the requestor, which is a better practice than placing return information on the contract.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Contracts should be designed for use by a single Integration Mediator Function.	IL2.0	Integration Layer
Problem Statement	Should we create more generic, broader contracts that can be leveraged by several Integration Mediators, or should we be specific about the data needs for a single service and limit the usability of the contract to a single Integration Mediator Function?		
Assumptions	• Optional information increases the complexity of the contract. Makes it more complicated to document and explain for use by end users. • Optional information increases the likelihood of implementation errors, thus increasing the complexity of the contract validation that needs to be		

	performed. • Case statements to support the business rules of optional information are very susceptible to change, and in many situations the case statements for different usage scenarios will diverge over time, making the logic more complicated and more difficult to maintain. • Optional information cannot be eliminated for all cases. • Very specific contracts that limit or eliminate the optional data reduce the potential for their reuse, thus increasing the number of classes that need to be maintained in the solution. • Unique contracts per Integration Mediator eliminate the possibility of a shared contract having to be split into two in the future when the information rules for the various Integration Mediators using the contract diverge. By eliminating this possibility, the service signatures will be more stable over time, thus improving the maintainability of the business objects that leverage the services.
Motivation	Simplify the usability of the Integration Mediators for the users that will have to design to the Integration Mediator Function. Provide a solution that provides the most stable Integration Mediator Functions for the users to minimize the impact of change to the user throughout the life of the solution.
Alternatives	1. Leverage a single Integration Contract that contains accessor methods to all the information that will be needed by any Integration Mediator Function in the system. 2. Share Integration Contracts across Integration Mediator Functionss that have similar purposes to reduce the number of contracts that need to be maintained and take advantage of reuse. **3. Define an Integration Contract for each Integration Mediator Function.**
Decision	Option 1 was not selected primarily because the contract implementation would be excessively large, which would be disadvantageous in a distributed environment. The contract would be complicated for users to design to, because the specific data needs for the service they need to leverage aren't obvious in the definition of the class. The documentation required to describe the data needs per Integration Mediator would be difficult to develop, cumbersome to read through for the user, and hard to maintain as service needs change and as services are added. Option 2 was not selected due to the risk that over time the information needs of even similar Integration Mediator Functions could change; thus, requiring the contracts to be broken up, which would cause the service method signatures to change. The potential negative impact of this change effect to the various users of the Integration Mediator Fucntions are greater than the benefits that will be reaped by reducing the number of contracts in the system through reuse of the contracts. **Option 3 was selected** because it provides the most straight forward definition of the information needs of an Integration Mediator Function to its' users. Since the contract is used by only one Integration Mediator Function, optional data will be significantly reduced. Option 3 also eliminates the possibility that over time an Integration Mediator Function's signature will change due to its contract type changing. Since only one Integration Mediator Function uses a particular contract, the contract content

	can change to support the changing information needs of that Integration Mediator Function. There is no risk that those information needs will diverge from another Integration Mediators Function needs, since no other Integration Mediator Function leverages the contract.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Contracts encapsulate the information needs of the Integration Mediator Function and are abstract implementations of the Interface pattern.	IL3.0	Integration Layer
Problem Statement	How can the requests for information required by the Integration Mediator Function to perform its service be defined, while providing maximum flexibility in the implementation for the requestors?		
Assumptions	• Integration Mediator Functions will be independent of the knowledge of those requesting its service. • Implementation flexibility is critical to allow users to take advantage of various application distribution architectures and technologies.		
Motivation	Design an Integration Mediator Function that can be leveraged across requestors; thus, 1) ensuring consistency of response across various requestors and 2) reducing maintenance by eliminating redundant silos of integration logic.		
Alternatives	1. Create a concrete class that encapsulates the requests for information and place an instance of this class as the parameter for the Integration Mediator Function. **2. Leverage the interface pattern to create an abstract definition of the requests for information that the Integration Mediator Function expects answered. Place an instance of this interface as the parameter for the Integration Mediator Function.**		
Decision	Option 2 was selected, because it allows for implementation flexibility, thus better addressing the second assumption.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Contract methods can return native types and simple object types (e.g., String) only	IL4.0	Integration Layer
Problem Statement	We must ensure that we do not create a presumed dependency for order of execution of SOA Services accessing the Integration Adapter layer.		
Assumptions	• Cannot define the Integration Contract methods to require a return type that would only be accessible by executing another request to the Integration Layer first.		
Motivation	Eliminating an order of operation prerequisite for the Integration Mediator Functions is essential to delivering a flexible SOA architecture.		
Alternatives	**3. Allow the Integration Contract methods to return native types and simple object types only.** 4. Allow the Integration Contract methods to return native types and simple object types (Alternative 1), and also allow them to return Immutable Data Object Interfaces.		
Decision	Option 1 was selected, because the only way for a instantiator of a Integration Contract to have access to an Immutable Data Object Interface would be to require another request to the Integration Mediators be executed first. Such a dependency on the order of operation should be encapsulated within the Integration Mediator layer to eliminate potential misuse and simplify use of the		

	Integration Mediators for the users.

Integration Mediator

Architectural Decisions:

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	All logical units of work to manage the integration of disparate back-end technology and any physical transactions must be started, executed, and closed within the execution of the Integration Mediator Functions	IL5.0	Integration Layer
Problem Statement	Where should transaction management occur within the layered architecture?		
Assumptions	• Knowledge of the underlying enterprise and database products/technologies cannot pass beyond the Integration Adapter Layer.		
Motivation	Encapsulate the complexity of coordinating integration technologies to retrieve data from the user of the Integration Mediator.		
Alternatives	**1. The Integration Mediator Function is responsible for managing the logical unit of work.** 2. The Data Object is responsible for managing the logical unit of work. 3. The Integration Adapter is responsible for managing the logical unit of work.		
Decision	**Option 1 was selected**: Was selected because a logical unit of work may operate across multiple unrelated Data Objects, and across different integration technologies (thus across multiple Integration Adapters. A Data Object can manage its own sub-logical unit of work within the logical unit of work of the Integration Mediator Function.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Mediator Functions must take as their only parameter an Integration Contract	IL6.0	Integration Layer
Problem Statement	Provide a simple, consistent API for communicating with the Integration Mediator Function that will reduce the impact of changing information needs of the Integration Mediator Function.		
Assumptions	• Simplifying and stabilizing the Integration Mediator Function signature improves communication between the requestor and the Ingration Mediator owner, which has a positive affect on integration quality and the possibility of parallel development.		
Motivation	Reduce maintenance costs by simplifying the architecture and decoupling the layers making them less susceptible to the impacts of change.		
Alternatives	**1. Pass all information to the Integration Mediator Function in an abstract interface of an Integration Contract that encapsulates all its' information needs.** 2. Expose each piece of information needed by the Integration Mediator Function as a separate parameter on the method signature. 3. Pass all information to the Integration Mediator Function in an concrete implementation of an Integration Contract that encapsulates all its' information needs.		
Decision	**Option 1 was selected**. Making sure that the parameter type is an abstract implementation of the Interface pattern, rather than a specific, concrete class type is critical to allow the requestors flexibility in how they implement the contract.		

	Option 2 was not selected because it exposes all the information that the Business Mediator Function will use in the method signature. When the information needs of the Business Mediator Function change, the requestor will need to modify the method that it calls to the updated method signature. Implementation is complicated, as well, as the developer must confirm that the order of the attributes passed is consistent with the order specified in the method signature.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Mediator Functions can return one of five values: void, an Immutable Data Object Interface, a collection of Immutable Data Object Interfaces, a native type, a collection of native types	IL7.0	Integration Layer
Problem Statement	What is the best way for the Integration Mediator Function to respond to the requestor with respect to separation of concern and flexibility to future requirements changes?		
Assumptions	• The requestor should know as little as possible, internally, about the Integration Layer.		
Motivation	Ensure consistency of return information from the Integration Mediator Function, and minimize the internal knowledge of concrete entities in the Integration Layer.		
Alternatives	1. Have the Integration Mediator Function return native types, simple Objects (e.g., String), or Data Objects to the requestor. **2. Have the Integration Mediator Function return native types, simple objects (e.g., String) or Immutable Data Object Interfaces to the Business Object.**		
Decision	Option 1 was not selected, because it would result in direct coupling to a concrete object in the Integration Layer, thus providing zero flexibility for change in the implementation. Having the Provider Router deal directly with a concrete class rather than an abstract implementation of the Interface pattern limits the ability of the Integration Layer to change as the needs of the business evolve. This solution will not allow the Integration Adapter to change without significant change impact in the Provider Router. **Option 2 was selected**, because it guarantees separation of concern and adheres to architecture layering best practices. Additionally, it provides greater flexibility in the implementation than option 2, as it returns an abstract implementation of the Interface pattern (Immutable Data Object Interface) rather than a concrete Data Object.		

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Mediator Functions can pass reference to the Integration Contract to objects within the Integration Layer.	IL8.0	Integration Layer
Problem Statement	Should the Integration Mediator Function be allowed to pass the Integration Contract to the Data Objects, since the Integration Contract represents the input data that the Data Objects require to communicate with the integration technology (e.g., databases, middleware, b2b APIs)?		
Assumptions	• Integration Contracts are designed for consumption by the Integration Mediator to fulfill that service's information requirements. • Forcing a Data Object to use a particular contract may limit its' reusability		

	among Integration Mediators and across the organization. • Integration Contracts can contain information that an Integration Mediator will use across several Data Object requests.
Motivation	Design an architecture that is flexible to changing business needs.
Alternatives	**1. Allow the Integration Mediator Function to forward the Integration Contract to the Data Object or helper classes in the Integration Adapter. The Data Object or helper classes can then pick the data that they need.** 2. Extract the content from the Integration Contract via the Integration Mediator Function. Allow the Integration Mediator Function to forward the required information to the appropriate Data Object to execute the request. The Integration Mediator Function drops reference to the Integration Contract at the conclusion of its execution. 3. Extract the content from the Integration Contract via the Integration Mediator Function. Have the Integration Mediator Function retrieve an instance of the appropriate Data Object or helper class and then have the Integration Mediator Function place the appropriate information on that object using its setter methods.
Decision	Option 2 was not selected for two related reasons: 1) option 2 is inefficient from a design, development, and run time perspective when compared to the option of passing the contract, and 2) option 2 would result in method signatures with a large number of parameters. Methods with a large number of parameters are cumbersome to implement and as the data needs change for the method the parameter list will change causing a change impact to any requestor. Option 3 was not selected, because it too is inefficient, and by making the Integration Mediator Function instantiate and set the data on the Data Object the Integration Mediator becomes unnecessarily coupled to the Data Object. The result of which will be increased maintenance due to the impacts of change. **Option 1 was selected**, because it allows us to keep the parameter list on the Data Objects simple and does not tightly couple the Integration Mediator Function to the Data Object. Additionally, this solution is the most efficient approach. In the event that multiple Integration Mediator Functions would need to use the same Data Object, there would be a need for that Data Object to provide separate methods that have as a parameter one of the expected contracts (e.g., retrieve(ContractA) and retrieve(ContractB).

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	An Integration Mediator Function can leverage other Integration Mediator Functions to delivery its service.	IL9.0	Integration Layer
Problem Statement	Should Integration Mediator Functions be self-contained and depend on no external Integration Mediator Function, or should a Integration Mediator Function reuse operations provided on other Integration Mediators.		
Assumptions	• Objects in the same layer can have public access to one another without violating the best practices of layered architecture. • Object oriented best practices propose the reuse of functions rather than the		

	duplicating the lines of code. • Duplicated lines of code increase complexity and hurt maintainability. • Reusing services from another object couple you to that object. As coupling increases between objects, so does the potential impact of change to the calling object when the called object changes.
Motivation	Develop Integration Mediators that are easy to maintain and resilient to the impact of change.
Alternatives	**1. Allow Integration Mediator Functions to leverage other Integration Mediator Functions.** 2. Don't allow Integration Mediator Functions to leverage other Integration Mediator Functions to ensure they remain self-contained. 3. Allow Integration Mediator Functions to leverage Integration Mediator Functions on its Integration Mediator, only.
Decision	Option 2 was not selected, because it results in duplicating lines of code to perform the same function. This is inefficient from a development and maintenance standpoint, and poor OO design and implementation. It does provide the maximum decoupling across the Integration Mediators, but the expense is too great. Option 3 provides a little more flexibility in reusing services, while retaining the same degree of decoupling across Integration Mediators. Limiting the reuse of Integration Mediator Functions only to within the Integration Mediator still has the same weaknesses as Option 2, therefore it was not selected. **Option 1 was selected**, because it follows the recommended object oriented best practice of leveraging services within the same layer to improve development and maintenance efficiency, and reduce the complexity of the solution by isolating identical functionality to a shared service. Option 1 does increase the potential of coupling amongst the Integration Mediators, which can have a negative effect on the impacts of change to the solution, however, the impact can be isolated within the Integration Mediator itself in most cases. Therefore, the potential of the change impact reaching the Provider Router is no greater with option 1 than option 2 or 3.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Integration Mediators are not intended to have a single Integration Mediator Function	IL10.0	Integration Layer
Problem Statement	What should an Integration Mediator class contain? Should it have only a single Integration Mediator Function?		
Assumptions	• Object metric best practices suggest that throughout a project, classes should have an average of 10-15 methods, with an average of 15 lines of code per method.		
Motivation	Develop Integration Mediators that logically group services of the application to simplify the usage of the system and correlate with the real-world business process relationships.		
Alternatives	1. An Integration Mediator is defined for each Integration Mediator Function. **2. An Integration Mediator consists of a group of related Integration Mediator Functions from a business perspective.** 3. An Integration Mediator consists of Integration Mediator Functions for		

	a single Data Object or hierarchy of Data Objects.
Decision	Option 1 was not selected, because it creates an unnecessarily large number of classes and does not result in groupings of like services. This option is less efficient from an implementation/maintenance standpoint, as it tends to increase the lines of code. Option 3 was not selected either. Integration Mediators should be looked at as groupings of like services, rather than grouping of services for a Data Object. The concern with this approach is that taking this viewpoint may lead to the Integration Mediator doing nothing more than method chaining to the Data Model, which results in integration processing logic transferring to the Data Model. As this process information transfers, the coupling within the Data Model will increase, which will make it more brittle. **Option 2 was selected**, because it results in Integration Mediator Functions that are bundled similar to the way the business views their primary processes being bundled. It results in service-oriented objects that focus on managing the processes and directing the objects to work together to deliver the services. It does not result in an Integration Mediator per Integration Mediator Function, nor a single Integration Mediator for all the Integration Mediator Functions.

Contract Validator

Architectural Decisions:

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	**Prohibit the duplication of rules in the Contract Validator**	IL12.0	Integration Layer
Problem Statement	Should we allow enterprise business logic edits (e.g., legacy application edits) or database edits to be checked in the Contract Validator?		
Assumptions	• Significant performance improvements have not been observed by moving the Data edits closer to the UI. • Duplicating edits in the architecture results in maintenance consistency problems and greater number of defects.		
Motivation	The desire to identify business rule violations as quickly as possible in the system, results in the temptation to pull enterprise business rules and database edits closer to the user interface.		
Alternatives	1. Allow database edits (length edits, format edits, etc.) or enterprise business edits to be duplicated in the Contract Validator to detect errors earlier. **2. Prohibit the duplication of rules in the Contract Validator to avoid errors associated with keeping the duplicated rules in sync.**		
Decision	Option 1 provides for earlier detection for rules violations in the architecture; however, it introduces unacceptable risk of errors due to rules becoming inconsistent, and it creates added cost in maintaining the solution. Duplicated rules are difficult to track and keep consistent, therefore introducing the possibility for defects. Additionally, placing database or enterprise application specific edits in other layers of the solution increases the coupling of the solution to those underlying information sources and reducing the flexibility.		

	Option 2 was selected because it prevents the duplication of rules (e.g., database edits) and ensures the business logic of the application is decoupled from its underlying information sources (e.g., databases and enterprise apps). With the advent of good rules engines in the marketplace, there is now the possibility of adhering to Option 2, but achieving the earlier detection benefits sought in Option 1. By externalizing the rules to a Rules Engine, a solution's rules can remain unique (unduplicated), while being integrated into multiple different validation checkpoints. This approach does adhere to the architectural decision of Option 2, as it prevents duplication of rules.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Contract Validation should be managed by a support object that the Integration Mediator Functions leverage, rather than a function of the Integration Mediator itself.	IL13.0	Integration Layer
Problem Statement	Should contract validation be a function of the Integration Mediator Function or a separate support object within the application?		
Assumptions	• Integration Mediators may have several public service methods, which each require a different Integration Contract. Each of those contracts may need to be validated. • Object-oriented metrics best practices recommends that a application should have a class average of 10-15 methods per class, and 15 lines of code per method. Classes that greatly exceed these metrics should be reviewed for the possibility of redesign.		
Motivation	Provide a design recommendation for designing The Contract Validator that is efficient to develop/maintain, allows for reuse of validation services, and does not result in excessively large or small classes.		
Alternatives	1. Using method overloading, create a validate method on the Integration Mediator itself for each Integration Contract that needs to be validated. **2. Create a centralized Contract Validator that contains a validate method for each Integration Contract that needs to be validated for all the Integration Mediators in the solution. Leverage method overloading to ensure a consistent approach to accessing validation, thus simplifying the architecture.** 3. Create a Contract Validator per Integration Contract that needs to be validated. The object will provide a single validate method. 4. Define the validation of the Integration Contract within the Integration Contract itself.		
Decision	This architectural decision doesn't have one right answer. Options 1, 2, and 3 are all acceptable and have their own inherent strengths and weaknesses. Of these options, option 2 is the preferred approach. When deciding on the approach to apply for a particular implementation, the architect must assess the strengths and weaknesses of each option and their relavence to the solution at hand. Option 1 has the benefit of keeping the validation encapsulated and private within the Integration Mediator class. Its' weakness is, given the first assumption above, this option tends to result in Integration Mediators that press the limits or exceed the 10-15 method best practice metrics of OO resulting in overly complex objects that are harder to maintain;		

	Option 2 is recommended. By centrally locating the validation to a separate Contract Validator, we define the validation for a contract once and any Integration Mediator Function can access the logic. Additionally, by removing the validation methods from the Integration Mediator, we help to avoid the probability that the Integration Mediators will exceed the method metrics suggestion of 10-15 methods per class. The benefit of accomplishing this is that we now have two types of focused objects: Integration Mediators that focus on conducting various objects to deliver an integration process and validation objects that are used by the Integration Mediator and focus exclusively on validation of contracts. The weakness of option 2 is that we will end up with a validation class with far more methods than the metric best practice of 10-15 in order to validate the entire applications suite of Integration Contracts. Option 3 guarantees that the metrics best practice of 10-15 methods per class will not be exceeded, as was a concern of option 1 and 2. Option 3's weakness is that it results in a large number of classes which grow exponentially in numbers as the Integration Contracts increase in number. Extremely small objects, are inefficient to develop/maintain and should be candidates for redesign. Option 4 is not allowed, since the Integration Contract is an implementation of the Interface pattern. The Interface pattern allows you to define the method signatures required, but does not allow you to define the method implementation. The method implementation is left open and flexible to the implementor of the Integration Contract. What does that mean in layman's terms? It means that we could define a validate method on the Integration Contract interface, but we could not enforce the actual rules that were applied in doing the validation by the implementor (implementor in this case means that actual class that implements the Integration Contract interface, in this case the Process Router or another Integration Mediator Function). The end result… we would not be able to guarantee consistent rules applied in the validation process, across users of the Integration Mediator Function, which defeats the purpose of the validation. Taking this thought one step further, it has been recommended that we define the actual class that implements the Integration Contract interface, so that we could define and enforce a consistent validate process. This approach would only work if we changed the Integration Mediator Function to require a specific Integration Contract implementation class, rather than the Integration Contract interface. That would violate Architectural Decision – IA***4.0, Integration Mediator Functions must take as their only parameter an abstract implementation of the Interface pattern : an Integration Contract,*** defined in the Integration Mediator's Architectural Decisions above.

Immutable Data Object Interface

Architectural Decisions:

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	Immutable Data Object Interface will adhere to the guidelines of the Interface pattern and be implemented by it's Data Object	DM1.0	Model
Problem Statement	How can we create a view of the Data Model that can be returned by the Integration Mediator Function to the Provider Router, but does not tie the Provider Router to a specific implementation of the Data Model?		
Assumptions	• Implementation cannot provide access to methods that would change the state of the Data Model (e.g., setter methods).		
Motivation	Architect a solution that preserves MVC separation between the Provider Router and the Data Domain in the most efficient means possible.		
Alternatives	1. Create an implementation of the Proxy pattern to be returned from the Integration Mediator Function to the Provider Router. Attributes of the Data Object will be transferred to the lightweight, Proxy instance. **2. Create an implementation of the Interface pattern to be implemented by the Data Object. Allow the Interface to be returned by the Integration Mediator Function to the Provider Router.**		
Decision	Option 1 provides a lightweight picture of the Data Object that can be beneficial when deployed on a distributed environment. Option 1 also increase the number of classes that need to be developed/maintained, and the number of instances that need to be instantiated and managed during run time. Typically, there is not a lot of extraneous data that is on the Data Object that would not be available for the Business Object, so the Proxy itself is not much more lightweight than the original Data Object. Therefore, the cost of maintaining the additional classes and the performance degradation related to instantiating, populating, and garbage collecting the Proxy are not worth the lightweight benefits that might be delivered for a distributed application. **Option 2 was selected**. It provides better flexibility as the returned value is an Interface (an abstract definition of message signatures), rather than a concrete Data Object or Proxy. The interface actually represents a different view on the Data Object, which means no additional objects need to be instantiated and populated, thus improving performance. While Option 1 was not selected as the preferred approach, there may be situations where it is superior to Option 2, such as distributed architecture situations where the payload sent across the network can be significantly reduced via a Proxy. Through performance modeling, architects should assess their particular situation to determine which alternative is best for their situation.		

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	Immutable Data Object Interfaces are only allowed to contain getter methods and inquiry methods ("is…").	DM2.0	Data Model
Problem Statement	How can we create a view of the Data Model that can be returned by the Integration Mediator Function to the Provider Router, but does not tie the Provider Router to a specific implementation of the Data Model?		
Assumptions	• Implementation cannot provide access to methods that would change the state of the Data Model (e.g., setter methods).		

	• Information returned to the Provider Router regarding the Data Model should not include access to integration technology specific information (e.g., relational database, SOA service, operational system ties, etc.).
Motivation	Enable the Provider Router to be leveraged across different data models that leverage different integration technologies, without the need to change the Provider Router itself to accommodate the various data models.
Alternatives	1. Create an implementation of the Proxy pattern to be returned from the Integration Mediator to the Provider Router. Data about the Data Object will be transferred to the lightweight, Proxy instance. **2. Create an implementation of the Interface pattern to be implemented by the Data Object. Allow the Interface to be returned by the Integration Mediator Function to the Provider Router.**
Decision	Option 1 provides a lightweight picture of the Data Object that can be beneficial when deployed on a distributed environment. Option 1 also increase the number of classes that need to be developed/maintained, and the number of instances that need to be instantiated and managed during run time. Typically, there is not a lot of extraneous data that is on the Data Object that would not be available for the Provider Router, so the Proxy itself is not much more lightweight than the original Data Object. Therefore, the cost of maintaining the additional classes and the performance degradation related to instantiating, populating, and garbage collecting the Proxy are not worth the lightweight benefits that might be delivered for a distributed application. **Option 2 was selected**. It provides better flexibility as the returned value is an Interface (an abstract definition of message signatures), rather than a concrete Data Object or Proxy. The interface actually represents a different view on the Data Object, which means no additional objects need to be instantiated and populated, thus improving performance.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Data Object Interface getter methods can return native types, simple object types (e.g., String), or other Immutable Data Object Interfaces	DM3.0	Data Model
Problem Statement	How can we ensure that the Provider Routers don't become coupled to one implementation of the Data Model?		
Assumptions	• Provider Routers cannot have access to any object that can modify the state of the Data Model (e.g, the Data Object itself). • Provider Router cannot have access to any object that can provide it access to an object that can modify the state of the Data Model (e.g, objects that can return Data Objects). • Provider Router receives information about the Data Model layer via the Immutable Data Object Interface.		
Motivation	Provider Routers requests to change the state of the Data Model must be handled by the Integration Mediator Functions.		
Alternatives	**1. Immutable Data Object Interfaces can provide getter methods that only return native types, simple object types (e.g., String), and other Immutable Data Object Interfaces.** 2. Immutable Data Object Interface can provide getter methods that return		

	all types in option 1 and other Data Objects.
Decision	**Option 1 was selected**, because it ensures that the Provider Router will never gain access to the specific implementations of the Data Objects, but rather will integrate with the abstract Immutable Data Object Interfaces (which are implementations of the Interface Pattern). Option 2 was not selected, because it allows the Provider Router to gain access to the concrete implementation of the aggregate parts of the Immutable Data Object Interface. This tightly couples the Provider Router to that Data Model, therefore, allowing the Provider Router to directly change the state of the Data Model without going through the Integration Mediator Functions, a violation of the MVC framework.

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	Immutable Data Object Interfaces getter methods should not have any parameters.	DM4.0	Data Model
Problem Statement	Should Immutable Data Object Interface getter methods be allowed to take parameter input data?		
Assumptions	• Parameter input data is typically used to trigger a state change or update the internal data of the Data Model. • The Process Router should not be allowed to make state or data changes to the Data Model without going through the Integration Mediator Functions. • Ensuring the preservation of MVC between the Provider Router from the Data Model is crucial for the maintainability and flexibility of the solution.		
Motivation	Design an architecture that ensures adherence to the MVC framework between the Provider Router and the Data Model.		
Alternatives	1. Allow the getter methods of the Immutable Data Object Interface to have parameters in all situations. 2. Allow the getter methods of the Immutable Data Object Interface to have parameters in the situation where the parameter data is not used to trigger a state change in the Data Model, nor used to update the data of the Data Object. **3. Do not allow the getter methods of the Immutable Data Object Interface to have parameters.**		
Decision	Option 1 is not allowed, because it allows the Provider Router to make direct state changes on the Data Model, thus violating MVC. Option 1 would allow the Provider Router to change the state of the data model without going through the Integration Mediator. Option 2 was not selected, because it introduces too many “what if” scenarios, therefore, complicating the decision tree of what is and isn’t acceptable for the Immutable Data Object Interface. Goal is to define an architecture that is as user friendly as possilbe, and allowing Option 2 would not help us meet this objective. **Option 3 was selected**, because it ensures separation between the Provider Router and the Data Model, and does not introduce complex decision trees into the usability of the architecture.		

Data Object

Architectural Decisions:

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	Aggregate Data Objects hold reference to the Immutable Data Object Interface of parts (can cast reference to its Data Object temporarily within the execution of its methods, when needed.)	DM5.0	Data Model
Problem Statement	Object oriented solutions are founded on the concept of proper use of aggregation. However, even appropriate use of aggregation between Data Objects increases the coupling in the system and therefore impacts its' flexibility and maintainability. How can we architect a solution that supports aggregation, but provides flexibility in its implementation?		
Assumptions	• Aggregate whole, Data Objects have a need to modify the state of their parts in specific situations. • Of the solutions developed using R4SC, we have observed that the need to access the Data Object parts is limited.		
Motivation	Reducing the coupling between the specific implementations of the Data Objects within a persistence model will reduce the impact of change amongst aggregate whole Data Objects when their parts change. Net improvement in the flexibility and maintainability of the solutions built on R4SC.		
Alternatives	1. Aggregate whole Data Objects hold onto their parts' Data Object reference. **2. Aggregate whole Data Objects hold onto their parts' Immutable Data Object Interface reference, and have the option at any time to cast that reference to its Data Object temporarily within the execution of a method.**		
Decision	Option 1 was not chosen because it increases the coupling between the concrete implementation of the other Data Objects in the data model. This makes the solutions less flexible and more susceptible to the impacts of change. **Option 2 was selected**, because it does not result in the direct coupling of an aggregate whole Data Object to the concrete implementation of its' Data Object parts. Rather, it couples the aggregate whole Data Object to a more flexible, abstract definition of its parts – the Immutable Data Object Interface. Since the Immutable Data Object Interface only defines the method signatures that are available, while allowing for the specific implementation of the interface to vary this approach is far more flexible than option 1. Additionally, since the Immutable Data Object Interface isolates the user from how the Data Object specifically implements its' messages, changes to the Data Object part can be better isolated, thus reducing the impact of change to the aggregate whole Data Object.		

Subject Area	R4SC	AD ID#	App Area
Architectural Decision	Data Objects must know how to instantiate, retrieve, populate, and persist themselves	DM6.0	Data Model
Problem Statement	Who should have the responsibility to instantiate and populate a Data Object? A		

	common implementation weakness in the development of object oriented solutions is in the external management of an objects lifecycle. We have repeatedly observed solutions in which one object (Object A) performs the task of instantiating another objects (Object B) instances, retrieving that instances (Object B's) information from persistent storage, and decomposing the information from persistence into the other object's (Object B) attributes through the use of Object B's setters. The result is the increase in the coupling of Object A to other, inappropriate objects in the system. Object B becomes weak, because it doesn't know how to retrieve, populate, and persist itself; therefore, it is not a true Data Object.
Assumptions	• Data Objects need to know how to instantiate, retrieve, populate, and persist themselves. • Data Objects can accomplish these tasks through the use of helper classes, but they are responsible for managing the process and isolating other, non-helper objects from the knowledge of how this is done. • As coupling increases defects and cost of maintenance increase, and flexibility of the solution decreases. All negative affects on the solution. • Solutions that don't adhere to assumption 1 have duplicate process logic and unnecessary increased lines of code.
Motivation	Development of solutions that are more flexible to change and, subsequently, have a lower cost of ownership.
Alternatives	**1. Data Objects define public service methods on themselves that others can call to retrieve, populate, and persist.** 2. Aggregate whole Data Objects are responsible for retrieving, populating, and persisting their part Data Objects. This knowledge is within the aggregate whole. The aggregate part is not aware of how to perform these tasks. A common implementation weakness in the development of object oriented solutions is in the external management of an object's lifecycle. We have repeatedly observed solutions in which one object (Object A) performs the task of instantiating another object's (Object B) instances, retrieving that instance's (Object B's) information from persistent storage, and decomposing the information from persistence into the other object's (Object B) attributes through the use of Object B's setters. The result is the increase in the coupling of Object A to other, inappropriate objects in the system. In effect, Object A becomes too stroing, and Object B becomes too weak, because it doesn't know how to retrieve, populate, and persist itself, therefore it is not a true Data Object.
Decision	Option 2 is not allowed, because it results in tight coupling of aggregate whole Data Objects to its aggregate part Data Objects, and removes essential information that should be internal to the part Data Objects. The increased coupling results in solutions that are brittle to change. Additionally, by removing information that the part Data Object should know internally, we have introduced a situation where multiple users of the aggregate part Data Object will have to duplicate the logic that was externalized from it, thus increasing the lines of code of the solution (e.g., if a Data Object doesn't know how to populate itself and two aggregate whole Data Objects use this Data Object as an aggregate part, both aggregate whole objects will need to implement the logic to populate the part).

Subject Area	**R4SC Service Component Pattern**	**AD ID#**	**App Area**
Architectural Decision	Instance creation and management of the Data	DM7.0	Data Model

	Object must be managed through a factory method.		
Problem Statement	Provide guidance for the instantiation and instance management of Data Objects to best meet the needs of a solution throughout its lifespan.		
Assumptions	• Tailoring of instantiation strategies is necessary to address hot spots during performance testing. • It is difficult to anticipate your hot spots proactively, so a good architecture provides flexibility to make changes to address the hot spots reactively. • It is essential to be able to respond to these changes rapidly, as stress/performance testing is often in the critical path to delivery.		
Motivation	Accelerated time to market through flexibility. A solution that delivers cost savings and a positive ROI over the life of the project. We need to make sure that the solution is not more expensive to implement than the change impact that would be expienced with a less encapsulated solution. Over the life of a solution it often becomes necessary to switch instantiation strategies to meet the changing non-functional demands of a solution. The architecture should encapsulates, from the requestors, the knowledge of whether an instance-based, pooling, or singleton instantiation strategy is being used.		
Alternatives	1. Where an instance-based strategy is deemed appropriate, allow the Data Object to expose the default instantiator as a public method to request instances. 2. Where a singleton strategy is deemed appropriate, allow the Data Object to expose a singleton method. **3. Instance creation and management of the Data Object must be managed through a factory method.**		
Decision	Option 1 locks the Data Object into an instance-based approach. If, at some point in the future, it becomes necessary to transition to a singleton or pooled instantion strategy, the requestors of the Data Object will need to change. In Java, this approach would have the requestor call the new() method on the Data Object. Option 2 locks the Data Object into a singleton strategy, resulting in the same change impact as option 1 if a shift to an instance-based or pooling strategy is required. In this approach, a singleton operation (e.g., getSingleton) would be defined as a class operation on the Data Object. The default instantiator would be made private to prevent anyone from creating new instances. An private attribute (singleton) would be defined on the Data Object to hold reference to the singleton instance. The getSingleton operation would first check to see if the singleton attribute was populated. If it was it would return reference to that instance, if not populated, it would call the private default instantiator to populate the singleton attribute and return the reference to the instance to the caller. In option 3, you would define a factory method (e.g., getInstance), which would encapsulate knowledge of how the instances are being managed. This approach would require the the default instantiator (e.g., new() in Java) would be defined as a private method. If an instance-based strategy was needed for the Data Object, the getInstance method would simply turn around and call the default instantiator returning the instance that was created. For a singleton situation, the getInstance method would check an attribute (e.g., singleton) to see if it was populated. If so, it would return the instance, if not it would call the default instantiator to create an instance, populate the singleton attribute with this		

	instance, and return the reference to the singleton instance to the requestor. Finally, if a pooling scenario is required, the getInstance method would check an attribute that holds the collection of instances (e.g., instancePool) and "check out" one of the instances to return to the requestor. If all the instances were in use the getInstance method could, if allowed, request a new instance. The getInstance method would be responsible for managing the pool size as defined by the architectural decisions for the project. So, option 3 addresses the issue of encapsulation and provides a very flexible approach, thus addressing the first motivation. The second issue was whether this approach would be cost prohibitive, which this approach is not. The cost of designing and implementing this approach versus option 1 or 2 is nominal in comparison. For that reason, option 3 is required.

Integration Adapter

Subject Area	**R4SC**	**AD ID#**	**App Area**
Architectural Decision	All physical transactions are must be started, executed, and closed within the execution of the Integration Adapter.	IA1.0	Integration Layer
Problem Statement	Which component should have specific knowledge of the integration technologies physical transaction model?		
Assumptions	• Knowledge of the integrated technologies should be limited to a single component to provide the greatest encapsulation.		
Motivation	Encapsulate the complexity of coordinating integration technologies to retrieve data from the user of the Integration Mediator.		
Alternatives	4. The Integration Mediator Function is responsible for managing the physical transactions. 5. The Data Object is responsible for managing the physical transactions. **6. The Integration Adapter is responsible for managing the physical transactions.**		
Decision	**Option 3 was selected**: Was selected because it is responsible for managing and encapsulating all the specific integration with the integration technologies. Option 1 and 2 were not selected, because this would result in additional components of the architecture becoming coupled to the integrated technologies. In the even that those technologies had to change, the impact of change would be greater as the number of entities coupled to them increase.		

R4SC Frequently Asked Questions

This section is intended to address the most common questions we have received about R4SC from practitioners in the field. If you do not see a discussion of your question, that simply means it hasn't been posed yet, so please go to the SOA Community site and browse the answers to new questions there. If your topic is not addressed, please post a new discussion thread. We're here to help. This section will be updated with each subsequent release to reflect the new questions posted on the SOA Community site.

What performance implications are created by the R4SC layers?

> As with any architecture there are trade-offs. A layered architecture is typically going to perform slower than a solution that has no layering. However, a non-layered architecture is going to be poor in comparison for flexibility and adaptability. In the R4SC, we have captured what we have found to be the most effective blend of all non-functional criteria of a solution. R4SC has been applied in a wide variety of situations where transaction volume was high, payload was large, and availability was essential, and the resulting applications have met client NFR expectations.
>
> It is essential that any project architect perform proper performance modeling and testing to ensure the performance NFR can be met. It is equally important to do this modeling to ensure you don't make unnecessary concessions with regards to other NFRs (maintainability, adaptability, etc.) to achieve greater performance when you are within your performance metrics. Too often, we as architects place too great a focus on performance and unnecessarily sacrifice other NFRs that are important to our clients.
>
> As an example, it is important from a maintainability and flexibility standpoint to apply model-view-controller best practices. Two common alternatives that help you accomplish this is the use of a Proxy or an Interface. In architectural decision BM1.0, *Immutable Interface will adhere to the guidelines of the Interface pattern and be implemented by it's Business Object*, these alternatives are assessed discussed for their strengths and weaknesses related to addressing performance and maintainability. Ultimately, the Immutable Interface was mandated to abide by the Interface pattern as it provided better performance during runtime and development.
>
> Performance was a key concern of the team that produced the R4SC, however, it was equally important to address the other NFRs architects encounter when delivering solutions, as well.

Do you use Process Mediator Functions similarly for composite and atomic processes?

> Yes, regardless of whether a service component process makes multiple or one primary call to the Domain Objects or other Process Mediator Functions, a Process Mediator Function is created to manage that process microflow logic. This ensures architectural consistency, which improves usability and understandability. Also over time, the process micro flow that starts out as a single call to a Domain Object or other Process Mediator Function may evolve to a composite set of calls as the business needs change. Always capturing this microflow as a Process Mediator Function ensures that the users of the service component process aren't impacted when an atomic process becomes a composite process over time.

Immutable Interface will support simple type will it preclude you from passing complex aggregate?

> This refers to architectural decision BM2.0, *Immutable Interface getter methods can return native types, simple object types (e.g., String), or Immutable Interfaces,* in the section Architectural Decisions: Service Component Pattern. Aggregate Immutable Interfaces may be returned to the requestor. Where possible, Simple objects may also return as an aggregate type (e.g., collections).

The Immutable Interface has an architectural decision that states it can only contain getter methods, can I also define methods that may not start with the work "get" but only return data about the object and does not allow for state change of the Domain Object?

> This refers to architectural decision BM1.0, *Immutable Interfaces are only allowed to contain getter methods,* in section Architectural Decisions: Service Component Pattern. The simple answer is yes. The architectural decision was defined as very specifically to address the typical scenario, that one would expose a function to receive information about a Domain Object with a "getter". There are situations, however, where for clarity sake a name other than a getter is more appropriate for clarity of the model. For example, on one of our projects, we had a scenario where a request to retrieve the balance of an account needed to be returned. The Account balance was simply a calculation of using a few attributes of the Account, and resulted in no change in state of the Account object. The client team did not want to call this function getBalance, rather opting for calculateBalance. This is not a violation of the architectural decision, as this method does not result in a state change of the Account object. The objective of this architectural decision is to ensure model-view-controller is preserved, by prohibiting a user to change the state of the Domain Object without going through a Process Mediator Function.