SOA Approach

DHCS
California Department of HealthCareServices

Service Oriented Architecture (SOA) Approach

July 22, 2016

Document Version History

Version	Date	Comments	By Whom
1.0	7/22/2016	First 2016 Completed Document	Tanya Rhodes
1.0	9/28/2016	Approved via TAB	DHCS Technical Architecture Board
1.0	10/3/2016	Approved as DHCS Standard Approach	Barney Gomez, CIO

Feedback, questions, or general comments? Send e-mail to:

DHCS EAO@dhcs.ca.gov

Contents

Figures

1. Introduction to SOA

This document describes an approach for service enabling the Department of Health Care Services (DHCS) current technical environment through the realization of a Service Oriented Architecture (SOA) approach. This will allow DHCS to employ a robust, scalable, and extensible SOA based solution on the Medicaid Information Technology Architecture (MITA) Framework to support enterprise-wide operational and business processes.

1.1 Origins

For many years, software developers have struggled with the problem of creating software that is easily to maintainable, extensible, and able to integrate with other systems. There have been various approaches to designing and writing software which has evolved over time:

- In the early stages of the software industry itself, programmers found that organizing code into modules made it easier to maintain and reuse discrete pieces of functionality. This led to the development of large libraries of code that are still used today.
- The next major revolution in software design was object orientation. Software was modeled on the lines of real world and virtual objects where an "object" had properties which defined its state and methods to change its state, which had some major advantages:
    - Objects could be extended as required without impacting another object.
    - Implementation details of an object were hidden from the users of that object, called "abstraction", providing a fixed entry and exit point for the object.
    - Greater ease of maintenance and extensibility; and this paradigm extended to distributed object systems where objects could reside on different machines across a network and talk to each other using various remote access protocols.
- The industry gradually realized that there was a huge market for pieces of functionality that could be plugged into software applications and customized for specific application needs. Thus, leading to the concept of software components.
- Service Orientation became the latest buzzword in the software industry. In this approach, software functionality is defined as a set of services.

While SOA grew out of the early 80's with the iterative and object based application definitions that were being developed at that time. Using the iterative process of building applications which forces the layering of applications, and ultimately the separation of the applications into discrete layers. This separation layers where the application is divided into an n-Tier architecture that has 3 distinctly layers that interact with each other (e.g., e.g., Presentation or User Interface (UI) layer, Business Logic layer and Data Access layer) leading to the use of orchestration which is the process of determining the path and order of processes.

1.2 What SOA Success Looks Like at DHCS

SOA is a methodology that requires change a fundamental change to an organizations way of thinking about IT and standard operating procedures. The most common pitfall for a SOA initiative is not the technical aspect, but the cultural. For SOA to succeed within an organization, all stakeholders, from the top executives and stakeholders to rank-and-file staff, must adopt a SOA mindset. This mindset begins with the following:

1. SOA is not a product that comes in a box
2. SOA is something you do, not something you buy
3. SOA is a way of life, not a project (it takes time to realize return on your investment)
4. SOA is about architecture and design, not technology
5. SOA is about sharing ideas and solutions which means you must address the resulting cultural tension and conflicts

To achieve a successful SOA environment at DHCS several goals must be met, beginning with:

- Establish the fundamentals of SOA as essential requirements for virtually all solutions (new or legacy modernization)
- All designs must be at Web scale, using cloud-first and mobile-first design patterns (even if cloud or mobile were not initially targeted)
- Enforce an event-driven interaction model for advanced scalability, extensibility, throughput and resilience.
- Coordinate software design efforts through a central SOA authority to ensure consistent quality across the organization, including central IT and line-of-business IT initiatives.
- Use enterprise level governance structures for SOA governance to ensure a consistent and coherent SOA model and implementation is used throughout DHCS and to drive organizational changes as SOA becomes the DHCS standard.

1.3 Benefits of Service Oriented Architecture (SOA)[1]

- **Platform Independence** – Since web services can be published and consumed across development and operating platforms, an enterprise can leverage its existing legacy applications that reside on different types of servers and build additional functionality without having to rebuild the entire thing. It also helps an organization to integrate its applications with its partners (e.g., Providers, Agency, State, Federal, etc.).

[1] Thomas Erl - Service-Oriented Architecture: Concepts, Technology & Design Prentice Hall (August 12, 2005)

- **Focused Developer Roles** – Since a service is a discrete implementation independent of other services, developers in charge of a service can focus completely on implementing and maintaining that services without having to worry about other services as long as the pre-defined contract is honored.
- **Location Transparency** – Web services are often published to a directory where consumers can find and request them. The advantage of this approach is that the Web service can change its location at any time without disrupting existing integration points and service calls. Consumers of the service will be able to locate the service through the directory.
- **Code Reuse** – Since SOA breaks down an application into small independent pieces of functionality, the services can be reused in multiple applications, thereby significantly reducing the cost of development.
- **Greater Testability** – Small, independent services are easier to test and debug than monolithic applications leading to more reliable software.
- **Parallel Development** – Since the services are independent of each other and contracts between services are pre-defined, the services can be developed in parallel, which considerably shortens the System Development Life Cycle.
- **Better scalability** – Since the location of a service does not matter, the service can be transparently moved to a more powerful server to service more consumers if required. Also, there can be multiple instances of the service running on different servers. This increases scalability.
- **Higher availability** – Service can replicate instances (called "spawning") to support as many consumer calls as it receives, then remove instances when not needed. This ability provides the "always available" requirement to meet high availability requirements which typically are around 99%.
- **Greater Extensibility –** Any of the services can be scaled up or down (via composite services) to meet changing business needs with having to create new solutions or applications

SOA is an architectural style and discipline that is defined by a set of design principles.[2]

1.4 SOA Principles

There are eight established principles of Service Orientation[3] which are used industry wide and have been adopted by DHCS:[2]

- Standardized Service Contract
- Service Loose Coupling
- Service Abstraction
- Service Reusability

[2] Thomas Erl - Service-Oriented Architecture: Concepts, Technology & Design Prentice Hall (August 12, 2005)

- Service Autonomy
- Service Statelessness
- Service Discoverability
- Service Composability

1.5 SOA Strategies

These principles are built into the application tier by focusing on the following strategies within the design, development, and deployment of a DHCS SOA[4]:

- Modularity – Aligned with MITA's Seven Standards and Conditions via the various domains (i.e. Provider, eligibility, claims, etc.) and the loose coupling between the domains. The SOA Approach begins the implementation of this by cutting the direct links between business domains and creating service connections. This allows each domain to act as a "module" providing DHCS with MITA compliance.
- Statelessness – Use of stateless service calls such as SOAP or Representational State Transfer (REST) Web service calls ensure that the service layer, and the business layer with it, are not coupled to the presentation layer. This practice of using statelessness should be propagated throughout the system using standards and governance. This process begins by having the DHCS domains use stateless service connections to communicate.
- Abstraction – Concealing the complexity behind implementation while keeping consumers un-impacted by implementation changes is a key goal of SOA. By implementing and enforcing SOA across DHCS domains will ensure the ability to insulate existing processes from changes in other processes at DHCS.
- Open Standards –Adhering to open standards at DHCS will foster flexibility, interoperability, and integrity of disparate applications and systems by encouraging the use and sharing of standards across all components in DHCS regardless of where they reside.
- Loose Coupling – Low coupling between components and services, and high coherence within components and modules- achieved through the implementation of service connections across DHCS Domains.
- Separation of Concern – Services and Components have a defined single responsibility that does not overlap any other service or component responsibility. Part of the SOA Approach is to merge similar services between domains into single enterprise level services removing repetition of processes, and by creating connections using stateless standard SOA exchanges.
- Interoperability – A very important goal of SOA is achieving interoperability with disparate applications and systems regardless of their implementation technology and platform. This is achieved through the abstraction of complexity away from the communication between DHCS domain components by creating connections using stateless standard SOA exchanges.

[4] Thomas Erl - Service-Oriented Architecture: Concepts, Technology & Design Prentice Hall (August 12, 2005)

- Flexibility – Easy adaptation and configuration lead to faster adoption of services and economy of scale. Each service or component should have only one function, so changes can be easily identified and isolated for quick change and testing. By creating services that help DHCS domains share communications, this allows any DHCS domain to access the same information without new coding or affecting the other domains.
- Reusability – Reusable services and components – DHCS services, by definition will be able to be used by any process within DHCS. So that a provider directory service is reused by any process needing provider information.
- Composability – Service orchestration and composition from reusable services and other composites. Various DHCS services could be used in a single process to achieve capabilities outside of a single service. i.e. you could have a composite service which calls member, provider, claims, and reporting services to send a detailed list of claims processed for a specific member.
- Scalability – One of the foundational principles used to scale application provisioning, deployment and performance, while meeting the ever increasing demand for services. Because services are isolated from other processes their calls can be isolated as well which allows for adding or reducing resources from a service or function as needed based on demand.

1.6 What is a service?

The specific definition of a service depends on who you are talking to and where they are in the SDLC. Just using the term "service" can be very misleading. In order to use the term correctly and avoid confusion, the following standard definitions (as approved by TAB, IAB, BAB) will be used in this document:

Business Service: This refers to the provision of business capabilities through logical groupings of operations. It may or may not denote an actual software unit, and may include several different software units or "Services".[5] This refers to a capability or process area such as "Claims" or "Provider".

Technical Service: This refers to a capability that is not specific to a particular business process, but provides support, infrastructure, and other capabilities for the business.[5] This would include areas like Security and Logging and may or may not be a software unit, or may include several different software units or "Services".

Consumer: The application or business process which uses the specific Service.[5] This is usually done as one software unit calling the Service using specific message exchange protocols.

Provider: The application or business process that creates and owns the service and makes it available for consumers.[5]

[5] Rosen, Lublinsky, Smith, & Balcer – Applied SOA, Service Oriented Architecture and Design Strategies Wiley Publishing(2010)

Data Service: This refers to the capability of data management. The ideal for data is to have a single source of reference, then using data services to provide the views that each individual process needs. [5]

Composite Service: This refers to a service which is made up of calls to other service as part of a workflow/process in order to perform a defined business function. [4]

Service: The use of the word service by itself usually refers to the actual software unit that is performing the actions defined to it. This is usually a single software unit, but could be a Composite Service. This is the definition that most of IT will use when referring to services. [4]

Web Service: This refers to specific services that are accessed via the web or HTTP. A service can be a Web Service or not, but a Web Service is always a service.[4]

A service is an implementation of a clearly defined business function that operates independent of the state of any other service. It has a well-defined set of platform-independent interfaces and operates through a pre-defined contract with the consumer of the service. Services are loosely coupled – a service need not know the technical details of another service in order to work with it – all interaction takes place through the interfaces.

Information between the consumer and the provider service are passed in XML, JSON, or other open formats over a variety of protocols. The main protocols that web services use today are SOAP (Simple Object Access Protocol) and REST (Representational State Transfer).

While REST uses the existing internet infrastructure (HTTP), SOAP is independent of the network layer and can use a variety of network protocols like HTTP, SMTP and the like.

2. DHCS Transition to SOA

SOA can, and will, fundamentally change the IT landscape and how applications work at DHCS. In fact, SOA will move DHCS from large siloed applications to smaller application modules. This means that applications won't be managed and maintained the way they are done today, but rather there will be services managed and owned by providers and the consumers of those services. This applies to data, as well as business processes since both should be accessed via services. The approach of taking an existing application and re-writing it to include services does not create a SOA environment, but rather a new siloed application with services. Projects to inject SOA should be based on a business function, not an application silo.[6]

To move towards this goal, DHCS must embrace a SOA approach that introduces services in layers.

- The first step in this process is to identify the various business domains which operate independently or are siloed from one another.
- Then the data exchanges between the business domains must be analyzed and modeled so a single exchange model is created. (This exchange model will act as the starting point for a department wide exchange standard.)
- Starting at the outer most edges of the various silos, wrap all exchanges in a service, consolidating similar or related processes into a single service with multiple options.
- The resulting services' contracts and routing will be through a DHCS service gateway.
- Work inwards until all business processes are re-constructed into services. This will leverage the existing business knowledge inherent in the existing code without disturbing business logic, while still allowing modernization and SOA to occur.

There are several steps to this path towards SOA which are detailed in the following sections.

[6] Mike Rollings - Changing Mindsets From Applications to Services, www.gartner.com(Dec 24, 2012)

2.1 Industry Best Practices[7]

SOA and services are the starting point for this strategy. You can't implement the new architectural approach described here if you haven't first understood, adopted and implemented SOA as you've designed new applications and integrated old ones. SOA is the starting point and the avenue to the new architecture for DHCS. The following eight steps, done in an iterative manner, repeated as needed, will Move DHCS forward to a fully modular, SOA enterprise:

- Start with Services – Separation of Concerns
- Virtualize the Services / Add SDAS
- Think Cloud for Back End Processes / Mobile for UI
- Add in Access Policies / Data Approach – ACID vs. BASE
- Privileged Data Access
- Add Control Intermediary
- Event Processing for and Eventful World
- The Future:
    - Context Discovery
    - In-Memory Data-Grids and DBMS

The following outlines a way to use the first two steps to move DHCS firmly into a SOA environment and lay the foundations for empowering DHCS to achieve a full SOA environment without losing control of the process and ensuring future growth.

2.2 Separation of Concerns

SOA is the starting point — the avenue to the new architecture models and practices needed to move DHCS from today to SOA with an eye on the cloud. DHCS would start by using the concept of separation of concerns to build service while striving to avoid locking together software that may need to evolve independently or be used differently.

The overall functionality of an application may be split up into a set of services based on some of the following criteria (or concerns):

■ **Discrete Function**. Don't combine functionalities that are useful in a stand-alone capacity (i.e., encapsulate deposit and withdrawal separately, then compose them for a transfer).

■ **Work Type**. Don't combine different types of processes, which may be optimized differently (i.e. encapsulate separately query, transaction, and process control functionalities).

[7] Natis & Altman - *The 12 Principles of Application Architecture for Digital Business and IoT*
http://www.gartnergroup.com

■ **Frequency of change**. Frequently changing logic should not be combined with logic that changes rarely, to avoid undue risk and cost of change.

■ **Scope of Visibility**. Some services are highly visible, or will be for public access, others only to be visible inside one application. This separation allows the optimization of interactions and proper granularity of services of different scopes.

■ **Required Quality of Service(QoS)**. Some services will run mission-critical or highly visible functionality, and others will be secondary. Using this distinction to separate the services will make the application environment more agile, scalable and cost effective.

■ **Security or Privacy**. Services that deal with sensitive data, such as personally identifiable information(PII), personal health information(PHI) and other HIPAA based requirements, should be separated from those that deal with public data. Separating them on this basis will strengthen the overall security of the application, while avoiding the costs and issues that can come from violating such constraints.

■ **Critical Dependencies**. Where possible, functionality that creates a critical dependency on a particular external resource (a certain data store, Web access, a cloud service) should be separated from the functionality that does not bear that dependency, to improve resilience of the overall environment.

Separation of concerns can economically add to the agility, security, scalability, resilience, openness for integration, reuse and extensibility of applications. It can also benefit the IT organization by supporting specialization of developers in areas of expertise, from process designers to business analysts and advanced performance analysts.

Application architects and designers should have all of these types of separable concerns in mind, but apply them carefully, gradually and while exercising a high degree of care and planning reflecting the realities of the application and the organization.

Working with both the business architects and users, as well as the technical architects is critical to make sure each of the separation is achieved in a way to benefit the entire SOA environment.[8]

2.2.1 Connection Points

Starting at the edges – the edge of each business process area or domain – the points at which siloed applications connect with one another – begins the process of separation of concerns across the whole enterprise. It sets the stage and creates a standard framework from which all future Existing to SOA

[8] Thomas Erl - *SOA: Principles of Service Design* Prentice Hall (August 12, 2007)

projects rest on. The "edges" are already evident in how DHCS refers to various domains, such as CA-MMIS, MIS-DSS, CAPMAN, etc.

2.2.1.1 Start with Existing Business logic

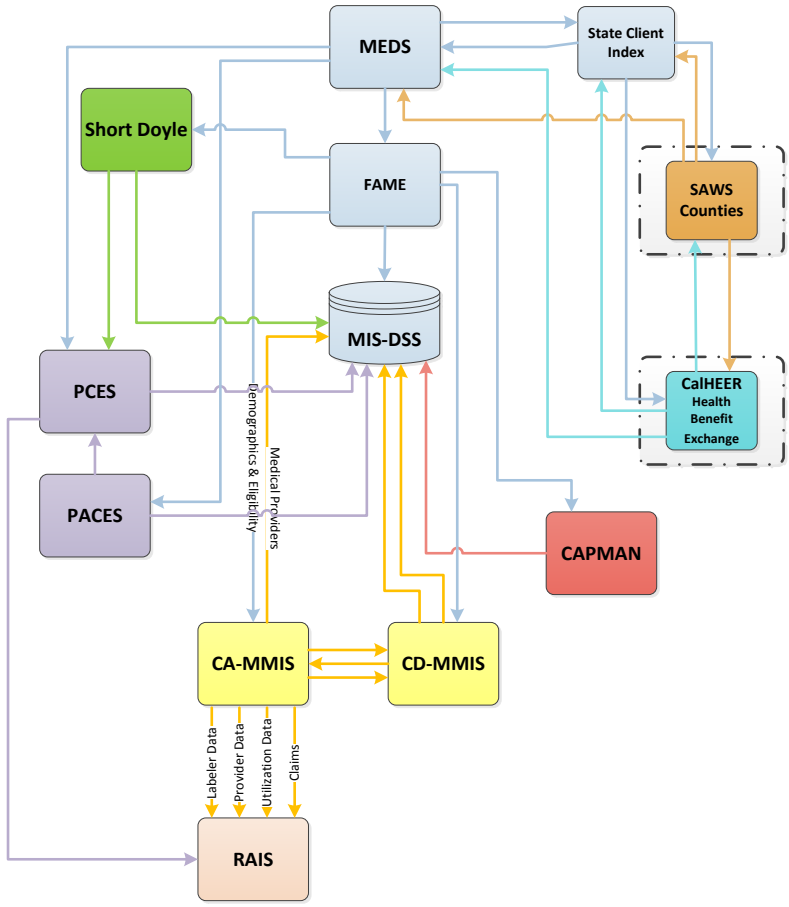

Figure 1 Contextual View of DHCS Systems

DHCS needs to know what the existing domains are and a general idea of how they exchange information between them. DHCS can start this process using a contextual view of the environment, as seen in Figure 1 Contextual View of DHCS Systems.

This gives a high-level view of each application area and its exchanges and can give us a picture of some of the services we might need to build. This can be used to do application rationalization or to decide which exchanges would be good initial service candidates based on predefined criteria such as, the value it brings to the enterprise.

Much of this work has already be done by DHCS using Sparx Enterprise Architect modeling tool to capture and model it (for example see Figure 2 - Provider Subsystem Data Dependencies Model). DHCS can leverage these models to better understand the application environment at and focus in on details

and patterns in the models that depict where information most often comes from. This help identify where potential service candidates might be early on in the process.

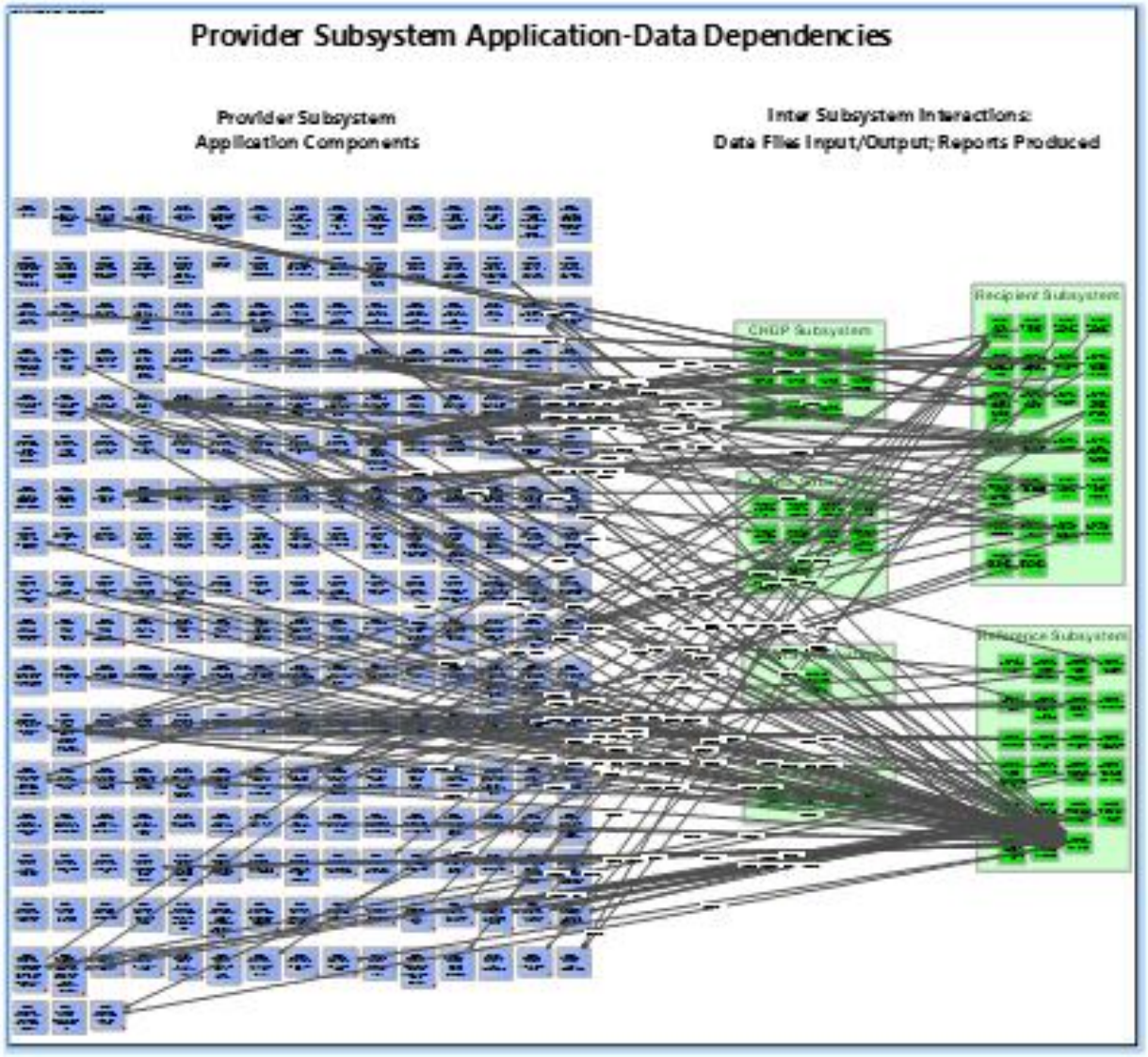

Figure 2 - Provider Subsystem Data Dependencies Model

2.2.1.2 Service Encapsulation

The diagram above (Figure 2 - Provider Subsystem Data Dependencies Model) is one of a set of diagrams which provides a detailed view of what processes, in which subsystems, call or exchange information from other processes. This Figure 2. focuses on the Provider system and shows where and how interactions with other systems occur. Simple analysis can identify where the service points are. The graphic presentation in the model gives us an eye view of the most obvious points where a service would be useful. This minimizes risk and scope by not making any changes to the business logic, but simply encapsulating it in a service wrapper.

The use of service wrappers to encapsulate existing application connection points and creating application level services is called service encapsulation, a concept that:

- Binds together the data and functions that manipulate the data.
- Keeps both sides of the service call safe from outside interference and misuse.
- Creates "isolated" functions. This enforces the separation of concerns that is critical for SOA to actually work.
- Provides "loose coupling" between components breaking the dependence of one application on another. As long as the interface point – the service data exchange – is not changed, then changes to either side of the service will not cause failure.
- Brings "coarse granularity" to services. By bringing related exchanges together into a single "source", we will begin to see the building of domain related services such as "Claims", "Eligibility", "Resource", "Provider", etc.

All of this brings the separation of concerns into the applications and exchanges at DHCS, providing the first big step in moving towards a SOA environment. For example, one of the obvious services from the above diagram would be a Reference Service with an option for MMIS tables. A service wrapper would be installed to manage calls to the Reference subsystem called Reference service. It would receive calls asking for information from the MMIS Tables process and would rend a request to that process for the request, then return that response to a service wrapper on the Provider side that would route the response to the requesting process. This would create a Provider service and a Reference service that can be used by anyone to retrieve information.

2.2.1.3 Archeology & Architecture, Adaption & Assembly[9]

To do this is a simple process (simple doesn't mean easy!) that should be undertaken by a small team of qualified people. To start with one team going after application domain and its connections would help create a foundational set of "how to" processes and build a framework for other teams to add to the effort to fully encapsulate the systems at DHCS. This "Tiger Team" approach would involve:

- Area of concern chosen and high level specs completed before team begins including standards, approach and architectural solution in place
- Small team of four with specific skills in services and service wrappers
- Team remains focused on ONE goal – no scope creep allowed
- Use of "Agile" approach to ensure speed and focus
- One deliverable – Services for Legacy functions

[9] Joe Bugajski - SWLegMod: A Framework for Legacy Software Modernization, www.gartner.com(Jan 19, 2010

Once the team is in place, they would need to analyze the legacy system and its exchanges to determine that all the input and output points have been identified and documented. They would also need to review all the behaviors that would be part of the service. This work is already underway for several of the subsystems through the EAO work on building Sparx models of the existing or "as-is" view of the architecture.

Next the team would design an architecture and create a solution that would provide the service encapsulation for the process. The team would perform the construction of the service in an Agile manner, using test-driven and sprint based approaches. The results of this would be documented including the methods and processes used to achieve the end result.

The last step prior to delivery is to test the new service against the original process to ensure that the results are the same and nothing was lost in the process of creating the services. Once the service is certified to be working correctly, final delivery of the service into production can occur.

2.2.1.4 What Encapsulation Gets DHCS

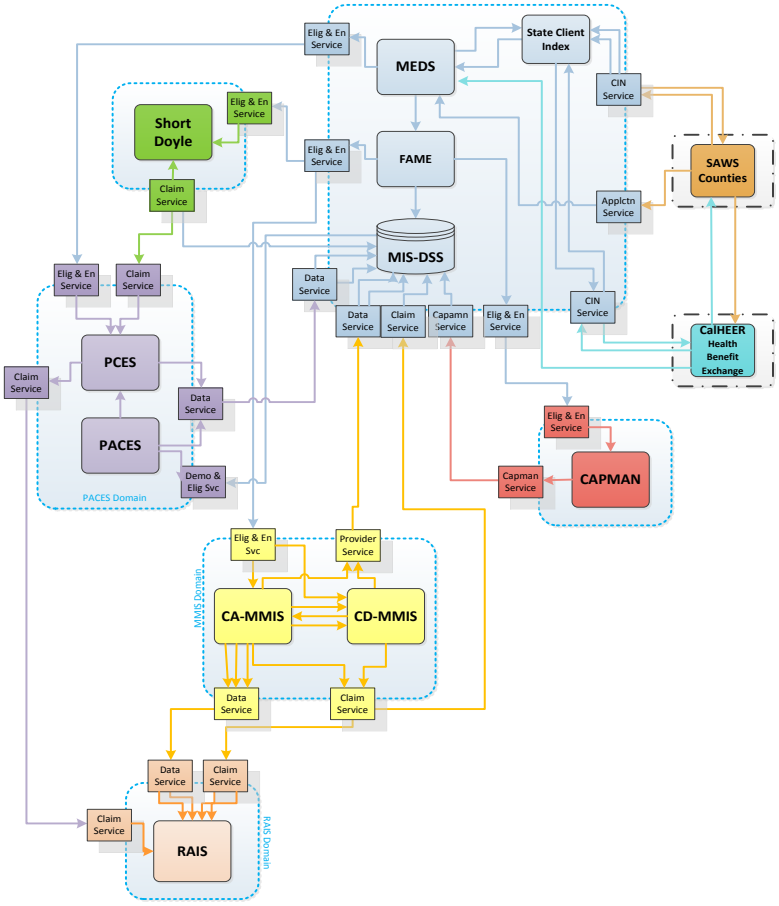

Figure 3 - Encapsulation Fully Realized

Encapsulation is a term that comes out of the object oriented movement and has become a backbone of design and architecture in IT. Service encapsulation is a SOA design pattern that came from a simple requirement:

How can solution logic be made available as a resource of the enterprise?

Service encapsulation is the philosophy of enclosing application logic with a uniformly defined interface and making these publicly available. The idea of complexity-hiding (abstraction), reuse, loosely coupling services that is part of encapsulation is what gives SOA its power. By encapsulating existing processes in a service wrapper, DHCS accomplishes several goals on the way to a full SOA environment:

- It takes the first step to full SOA enterprise environment
- Real, useful, enterprise level services up and running
- Beginning of "Separation of Concerns" throughout the enterprise
- "Think SOA" process has begun – crucial for ongoing and future SOA projects
- Builds interoperability between application domains
- Begins breaking applications into services – with full SOA there are no applications in the traditional sense – just orchestrated services performing business functions.
- Modularity introduced to application domains – a key step towards full SOA and MITA compliance
- Enterprise Level Architecture takes hold and the process of everyone using the same approaches and standards for providing business functions begins. This includes:
    - Enterprise Governance
    - Enterprise Wide Standards
    - Enterprise Wide Security
    - Enterprise Wide Data Exchange Requirements
    - Enterprise Wide Analytics

2.2.2 Virtualize the Services[10]

Once DHCS has a number of services in place, the application environment will gradually become dependent on a large number of services belonging to multiple applications or divisions. The proliferation of services creates a danger of losing control of the integrity, consistency and the quality of service (QoS) of the SOA environment.

To avoid this and to provide further support and governance around services, an intermediary infrastructure such as a service gateway needs to be utilized. DHCS should start simply with a Service Gateway to manage the service traffic, provide statistics and metrics, and AAA.

[10] Gary Olliffe – Choosing an API and SOA Governance Architecture, www.Gartner.com(Sept 2, 2015)

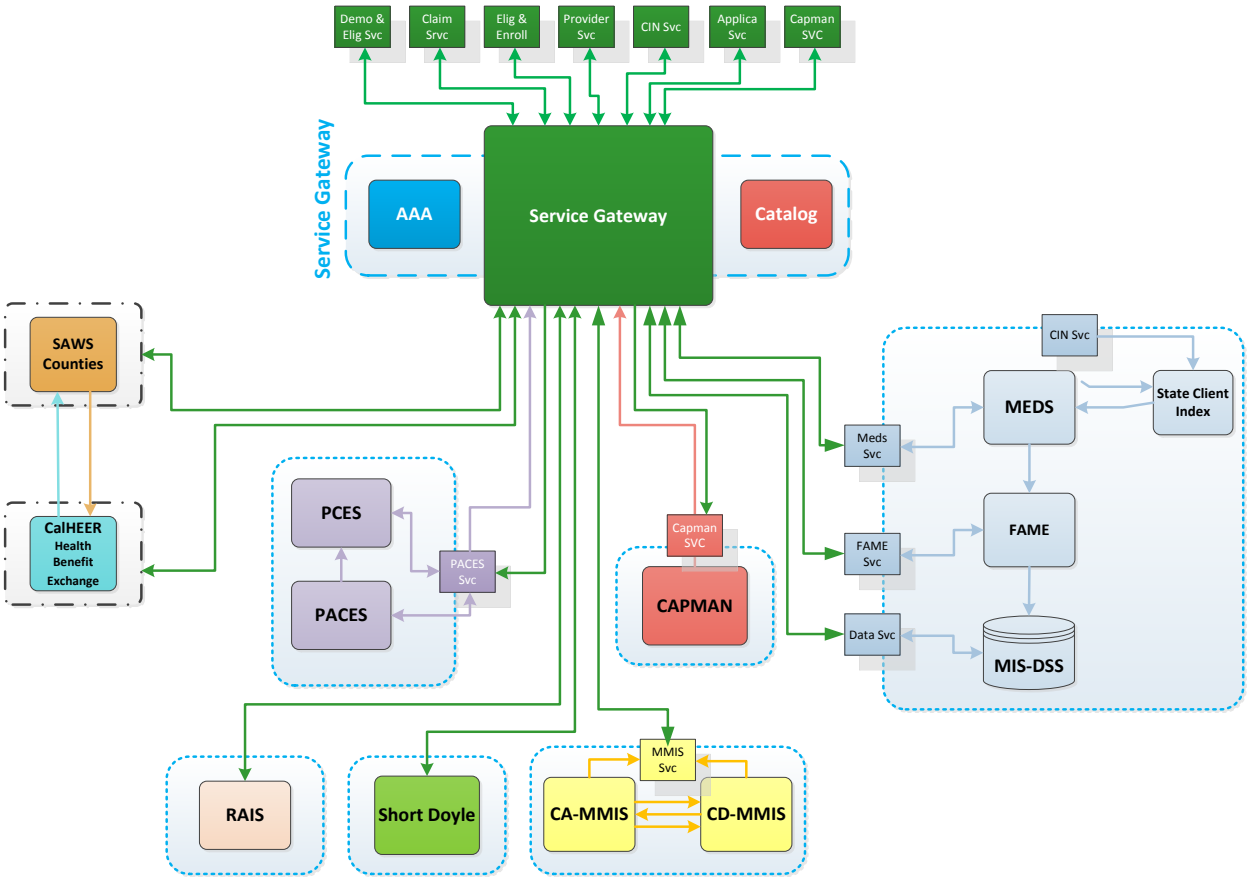

Figure 4 - Fully Realized Architecture with Service Gateway

The gateway exposes virtual (outer) software-defined application services (SDAS) as access points for the internal (inner) application services. It virtualizes the internal services and creates an opportunity to control their use. A service gateway can be used for tracking and accountability of service requests. The can also be used to add a variety of other valuable controls such as:

- Virtualization
- API Translation
- Protocol Translation
- Ad Hoc Virtual APIs
- Security
- Monitoring
- Optimization
- Routing
- Load balancing
- Integration

- Composition
- Orchestration
- Context Injection
- Scheduling
- Billing

The concept of an application is becoming uncertain – even obsolete, as many services are designed as composites accessing resources with multiple owners. Such uncertain ownership of transactions and their outcomes adds to the control challenges in SOA environment. The service virtualization gateway can further amplify the effect, but it also introduces the means to control the problem using some of the value add capabilities outline above.

2.3 Future State

Taking this approach to creating a SOA environment at DHCS will help establish the fundamentals of service-oriented architecture (SOA) as the design and build paradigm for all software projects — new or legacy modernization, at DHCS. This will provide the following benefits:

- Sets the foundation for a fully realized SOA environment at DHCS.
- Establishes enterprise level standards
- Establishes enterprise level policies
- Brings an Agile style process which is complete at each step
- Creates a framework for staff and vendors to work to, and for use with RFPs an RFIs, helping to eliminate duplication of products and application services which lowers cost.
- Begins changing analysis and design process to “think SOA”.
- Creates an extensible framework which can be used again and again to drill in SOA within DHCS.
- Builds a strong enterprise level unit for a consistent vision
- Engages a solid governance structure with processes and procedures for growing SOA at DHCS

The final result of the approach outlined in this document will be:

- Interoperability within the application domains, divisions, departments, and agencies.
- MITA Level 3 Featured Requirements Met – Service Oriented Architecture, Modularity,
- Helps meet CEAF 2.0 Reference Architecture:
  - Enterprise Application Integration (EAI), Version 1.0
  - Identity and Access Management (IdAM), Version 1.0
  - Service-Oriented Architecture (SOA), Version 1.0
  - Cloud Computing (CC), Version 1.0

Appendix A. SOA Definitions[11]

Appendix A.1. SOA Principles

There are eight established principles of Service Orientation:[12]

- Standardized Service Contract
- Service Loose Coupling
- Service Abstraction
- Service Reusability
- Service Autonomy
- Service Statelessness
- Service Discoverability
- Service Composability

Appendix A.2. SOA Strategies

These principles are built into the application tier by focusing on the following strategies within the design, development, and deployment of DHCS SOA:

- Modularity – Aligned with MITA's Seven Standards and Conditions
- Statelessness – Use of stateless service calls such as SOAP or Representational State Transfer (REST) Web service calls ensure that the service layer, and hence the business layer, are not coupled to the presentation layer. This practice of using statelessness should be propagated throughout the system
- Abstraction – Concealing the complexity behind implementation while keeping consumers un-impacted by implementation changes is a key goal of SOA.
- Open Standards – Principles of adhering to open standards foster flexibility, interoperability, and integrity of disparate applications and systems by encouraging the use and sharing of standards across all components in the system regardless of where they reside
- Loose Coupling – Low coupling between components and services, and high coherence within components and modules

[11] Thomas Erl - *SOA: Principles of Service Design* Prentice Hall (August 12, 2007)

- Separation of Concern – Services and Components have a defined single responsibility that does not overlap any other service or component responsibility
- Interoperability – A very important goal of SOA is achieving interoperability with disparate applications and systems regardless of their implementation technology and platform. This is achieved through the abstraction of complexity away from the communication between components
- Flexibility – Easy adaptation and configuration lead to faster adoption of services and economy of scale. Each service or component should have only one function, so changes can be easily identified and isolated for quick change and testing
- Reusability – Reusable services and components
- Composability – Service orchestration and composition from reusable services and other composites.
- Scalability – One of the foundational principles used to scale application provisioning, deployment and performance, while meeting the ever increasing demand for services

Appendix A.3. SOA Design Principles

To achieve the principles and strategies outlined above, a number of industry standard practices in SOA design principles need to be adopted and used to create DHCS applications. These will continue to be used to integrate Commercial Off The Shelf (COTS) products and changes needed for DHCS to be successfully integrated with all the various programs to create a coherent system of applications.

Abstraction

The principle of abstraction is normally used in the context of ensuring that a service is independent of a specific implementation and other detail. Services provide a high degree of abstraction from the service Implementation by using standards-based protocols rather than the native interfaces of the underlying technology. One of the principles of service orientation is to focus on what a service does, not how it does it.

For services to be effectively used, we use abstraction to:

- Remove implementation specific references from the service
- Hide data or behavior in the implementation that is specific only to the internal working of the implementation and not important to the service consumer
- Transform data types that are specific to the implementation technology
- Hide object behavior. Unlike object interfaces that might encapsulate the implementation we typically do not want to expose object behavior in the service that gives rise to inheritance, creates dependency on a specific object technology, or forces the service consumer into using an object oriented approach that may be inappropriate to the messaging style

For example:

checkDuplicateMember is a service provided in the Member module, which can:

- Verify if Exact Member is a Duplicate
- Verify a Suspect Member is a Duplicate

The purpose of this abstraction is threefold:

1. The service consumer does not have to know how to use all of the individual implementation-based services to check for duplicate member (Check Exact Duplicate Member/Check Suspect Duplicate Member).
2. From an agility perspective, the way in which the service provider chooses to configure those internal services/components can change over time without affecting the service consumer
3. The implementation-based services are still available to other developers in the project who require them to compose other business services

Generalization

The principle of generalization is to broaden the application of a service so that it can be used in a wider range of scenarios, including unexpected scenarios, removing the need to build a specific service for each new requirement.

The goals of service generalization:

- Separating common data and behavior from the specific, so that the common parts are more broadly applicable to a wider range of requirements and composed into many other services
- Including a wider breadth of data and behavior in the service than some individual service consumers might require, so that the service meets the need of broad range of service consumers without having to deliver them each a different service.

The goals of service generalization reflect a two-stage process to deliver comprehensive services, as follows:

1. First, decompose the service to find the fine grained parts both common and specialized;
2. Second, compose them back together to form a coarse grained service.

This can also be considered an aspect of granularity, which is discussed in section 1.3.6 Granularity.

An objective of using generalization to deliver a coarse-grained service is to reduce the number of services to expose and maintain. It can reduce the maintenance that is necessary when a service consumer uses the service in a new context, or realizes that they need some additional information from the service that they have not used before. With foresight, the generalized service can already provide this.

For example:

checkDuplicateMember is a service that performs:

1. checkExactDuplicateMember

2. checkSuspectDuplicateMember

From a generalization perspective, if the user wants to find only the Exact Duplicate Member, the checkDuplicateMemberService can be reused by parameterizing the service to fetch either exactDuplicate or suspectDuplicate.

Encapsulation

Services should not physically expose any implementation details or deployment details at their interface design. Well encapsulated services help adaptability by decoupling the service implementation characteristics and service deployment characteristics from client implementation. In circumstances where an implementation-specific or a deployment-specific characteristic needs to be changed, the client remains unaffected.

Service Composition/Aggregation

The functionality should be encapsulated as services at various levels of granularity and abstraction within a business. Services at one level of granularity or abstraction can be composed or aggregated to implement services at a higher level of granularity or abstraction, such as composing a business process from several business transaction services and exposing that process as a service itself.

The above mentioned principle is an important driving factor for the services *reuse* aspect.

There is no reason to limit the provision of the service to a single service at a fixed granularity. For example, a business service could be exposed as a single coarse grained service for use by third parties, and also as a set of fine grained services for internal use, or by closer business partners. This would provide a best of both worlds solution.

There are two ways that this ideal solution could be achieved. Either by simply providing a parallel service of the same facade, or by reusing the existing services and aggregating them into a new coarse grained service.

De-Coupling

A basic tenet of SOA is that the use of service interfaces and interoperable, location-transparent communication protocols means that services are loosely coupled with each other.

By loosely coupling services, we mean restricting what the consumer and provider of the services code know of each other to just the connection point of the service. That way, if a change is made to either the requester or provider aspect of a service that is decoupled, that change would be hidden from the other party using the service. That means that there should not be any need to change the service call

or connection. The service call or connection should be agnostic to any changes in the processes behind the end points of the service.

Interoperability

Interoperability plays an important role in SOA with platform/technology independence. The same can be achieved by using Web Services wherein the service contract defined and governed by a defined protocol (WSI standards).

A service component can be exposed as a Web Service binding, which helps achieve interoperability.

Granularity

The concept of granularity can mean several things in SOA:

- Level of abstraction of services - Is the service a high-level business process, a lower-level business sub-process or activity, or a very-low-level technical function?
- Granularity of Service Operations - How many operations are in a service, factors that determine which operations need to be collected in a service? - Granularity of service parameters. How is the input and output data of service operations expressed? SOA generally prefers a small number of large, structured parameters rather than a small number of primitive types.

Though it is generally preferred to have “Large Grained” services, some powerful counterexamples of successful, reusable, fine grained services exist. For example, getBalance is a very useful service, but is hardly large grained.

Service Granularity can be further defined at different levels as in:

- Technical Functions (Logger, Security)
- Business Functions (checkExactDuplicateMember)
- Business Transactions (CheckDuplicateMember)
- Business Processes (enrollMember)

Some amount of choreography, orchestration, or aggregation is required between each granularity level.

Large-grained interfaces (abstraction) help:

- Simplify coupling between processes – Where multiple systems that need to share the execution of a business process, large-grained interfaces come handy.
- Can be tolerant to changes – The use of large-grained service definitions inherently leads to more flexible systems

Granularity refers to the scope of functionality provided by a service. It is recommended that services be coarsely grained; however, there is no rule that services should be entirely coarse grained, or fine grained. The ideal is that they should be the right grain for the specific usage scenario.

Granularity Across Application Tiers

Services can be exposed at any application tier, and can be exposed directly from any database, object or component. Application packaging might expose Web services at a number of tiers within the application. There are different levels of granularity at each tier:

- **Business Objects**: Fine grained and not sufficiently abstracted from the implementation design
- **Database**: Database calls are also likely to be fine grained. Though some stored procedures might offer a coarser interface, as with components. However, either way this approach is likely to expose the internal database design which would not be desirable
- **Business Components**: If you have built components to package the business objects, then their interfaces are likely to be coarser grained, and better abstracted away from the fine grained object methods. There is no guarantee of this, as it depends on the design and purpose of the component
- **Business Process Components**: Likely to be coarse grained and a good match for external services, and a suitable level of abstraction that reflects some meaningful business service, not an internal interface

This document focuses on these principles, how they can be used strategically and tactically to create a an environment where service orientation is the norm, and how they provide:

- Component reuse through shared services
- Business logic abstraction from applications through business rules
- Technology to support SOA consistent with MITA
- COTS integration

The SOA Approach will define a way to satisfy the requirement to have SOA providing service access to business and technical processes within DHCS.

Appendix B. References

Version	Publication Date	Description
	11/4/2013	*Solution Path: Executing Your SOA* Initiativehttp://www.gartner.com/document/code/254072?ref=grbody&refval=2752017&latest=true
2nd Edition	11/2015	*Service-Oriented Architecture: Analysis and Design for Services and Microservices (2nd Edition)* by Thomas Erl
1st Edition	7/2007	*SOA: Principles of Service Design* by Thomas Erl
	2/2008	*SOA Design Patterns* by Thomas Erl
	8/2012	*SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST* by Thomas Erl
	9/2005	*A Portal May Be Your First Step to Leverage SOA* http://www.gartner.com
	1/2015	*The 12 Principles of Application Architecture for Digital Business and IoT* http://www.gartnergroup.com
	1/2010	*SWLegMod: A Framework for Legacy Software Modernization* http://www.gartnergroup.com
	7/2009	*Service Oriented Architecture Roadmap: Keeping on Track* by the Burton Group, http://www.gartnergroup.com
1.03	1/2011	*Architecture-driven Modernization:Abstract Syntax Tree Metamodel (ASTM)* Open Architecture Group http://adm.omg.org/
		http://www.soapatterns.org
		http://www.serviceorientation.com
1.3	8/2011	*Knowledge Discovery Metamodel (KDM)* Open Architecture Group http://www.omg.org/spec/KDM/1.3/PDF/