COMMON COMPONENT SPECIFICATION TECHNICAL DESIGN DOCUMENT: CCSSVCMGR01

SERVICE MANAGER

VERSION 1.0

TABLE OF CONTENTS

1. INTRODUCTION

The Common Component Specification (CCS) project is designed to simplify the development of modern business applications by defining the architecture and components of a modern application development framework for the OpenEdge platform. The CCS project identifies a set of specifications for the common components needed in developing modern business applications. This document describes the Service Manager as one of the required common components of the OpenEdge Application Architecture (OEAA).

The ability to program to abstractions (i.e. coding to a set of defined interfaces) is what allows the CCS to meet its goal of allowing interoperable components across frameworks and applications. The Service Manager is the infrastructure component that manages the relationship between the abstraction (an OO type name) and the implementing instance (a concrete, instantiable OO type name).

At its simplest, the Service Manager allows a component to request an instance of a (for example) `Ccs.Common.ILoggingManager` and receive a usable instance of a class that implements that interface without knowing or caring about the implementing type (e.g. `Application.Util.DbLogManager`) or how it was started or how long it will live for (transient or scoped to the life of the session or current request).

Purpose

The purpose of this document is to describe the required functionality of a Service Manager within the OpenEdge Application Architecture and define the API required by a CCS compliant Service Manager. This document can be referenced by application developers who are using a CCS compliant framework to understand the expected behaviour and functionality of a Service Manager. This document should be used by framework developers and implementers of CCS compliant frameworks as a definition of the minimum requirements and expected behaviour of a Service Manager.

Scope

The scope of this document is to define the required functionality of the Service Manager within the OpenEdge Application Architecture. The Service Manager MUST provide (object) references to the application for both framework/infrastructure and business/domain components. It is also responsible for instantiating these application services and managing their life cycle (startup and shutdown).

This document SHALL NOT define the Service Manager's management of the configuration data needed to perform its roles.

Definitions, Acronyms and Abbreviations

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (Bradner, S., Key words for use in RFCs to Indicate Requirement Levels , BCP 14, RFC 2119, March 1997. https://www.ietf.org/rfc/rfc2119.txt)

Term	***Definition***
CCS	Common Component Specification. Overview and links at https://github.com/progress/CCS
OERA	OpenEdge Reference Architecture *(Ormerod, 2006)*
OEAA	OpenEdge Application Architecture, as defined by the published CCS Architecture spec at https://github.com/progress/CCS/blob/master/Specs/CCSSpec-ARCH10.pdf *(Chase, Elwell & Smith, 2016)*
type name	An OO class, interface or enum name. Enums, interfaces and abstract classes are not instantiable.
marker interface	An interface whose mere presence indicates specific behaviour. Via https://en.wikipedia.org/wiki/Markerinterface pattern

References to external documents

Contents Overview

Section 1 is the introduction and includes a description of the Service Manager.
Section 2 provides the overview of the Service Manager.
Section 3 describes the interfaces and classes used by the Service Manager.

2. COMPONENT OVERVIEW

An application service is an object containing business and infrastructure functions that are either exposed to the client through (a) service interface(s) or by common reusable business and infrastructure functions used by other application components or services. Application services may be business components (which includes Business Entities, ~Tasks and ~Workflows as defined by the OpenEdge Reference Architecture / OERA) or infrastructure components (which includes Common Infrastructure components as defined by the OERA and/or the CCS-defined OpenEdge Application Architecture / OEAA).

The `Ccs.ServiceManager.IServiceManager` defines the primary component (the Service Manager) and API; a number of other interfaces are provided in support of it.

The Service Manager is classified as a Required Component of the OEAA. *(Chase, et al., 2016, p. 8)*

The Service Manager's API may be consumed by Service Interface components (for business components), by Business Components (for common infrastructures components not directly exposed by the Startup Manager; for other business components), by common infrastructure components (for other common infrastructure components, for business components, for dependency injection/resolution) as well as by other components not described in this specification.

Component Description

The Service Manager provides two categories of functionality: service name resolution and lifecycle management.

SERVICE NAME RESOLUTION

As mentioned in the Introduction, the Service Manager is the infrastructure component that manages the relationship between the abstraction (an OO type name) and the implementing instance (a concrete, instantiable OO type name).

The Service Manager provides a `getService` API to perform this name resolution. This API has two arguments, taken from:

1. The service name (REQUIRED). This is the primary selector (key) used to find an implementation. It MUST be an OOABL type reference (Progress.Lang.Class). An implementation SHOULD always use this type to validate the returned object instance's type. This type reference can refer to ***any*** OO type: it is typically an interface, but concrete and abstract classes can be used, as can enumerations.
2. A scope selector (OPTIONAL). The desired lifecycle scope of the requested service. The Service Manager MAY respect this parameter, or may choose to use its configuration data.
3. An alias (OPTIONAL). An application-/implementation-specific string. In certain cases an alias value as a second key value MAY be used to identify an implementing class. The Service Manager may use this information to make a better choice, given its internal configuration. The alias allows an implementation to provide special behaviour without requiring a full type name.

The normal `getService(App.GL.LedgerManager)` may return an object with a session scope, but an alternate, transient, instance may be obtained via a call using a specific scope: `getService(App.GL.LedgerManager, ITransientScope).`

An application may have a number of business entities that all implement the `Ccs.BusinessLogic.IBusinessEntity` interface with no other uniquely identifying interfaces.

A service interface may in this case request a service that implements IBusinessEntity interface and has an alias value of `eCustomer`.

RETURNED VALUE

The `getService` API is defined as returning a usable instance of `Progress.Lang.Object`; the actual type returned SHOULD be of the requested type when a type reference is used.

A "usable instance" is defined as an object instance that has been instantiated (non-null) and has had any initialisation performed (i.e. for types that implement the `IService` interface, the `initialize` method has been called).

Applications that use (persistent) procedure-based business logic services MUST return the procedure handle wrapped in an object, such as an instance of the `OpenEdge.Core.WidgetHandle` class provided by Progress Software. Extra care should be take to manage the life of the procedure handles since the AVM's garbage collection does not operate on handles.

SELECTORS-TO-IMPLEMENTATION MAPPING

The `getService` API SHOULD use some form of externalisable configuration data (like config files or database tables) to specify the selector-to-implementation mapping, although this is left entirely to the implementer (and so could be a CASE statement or REPLACE function).

An implementer MAY choose to use explicit mapping or an algorithm-based approach or an approach that favours convention over configuration.

Service type	***P.L. Class IsInterface***	***P.L. Class IsAbstract***	***P.L. Class IsEnum***	***Implementation ___ support***	***___ support alias***	***Comments***
Interface	YES	NO	NO	MUST	SHOULD	If an alias is used and there is no exact implementing class found in the configuration data, the implementation SHOULD try to resolve the selection without the alias.
Abstract class	NO	YES	NO	MUST	SHOULD	If an alias is used and there is no exact implementing class found in the configuration data, the implementation

Service type	*P.L. Class IsInterface*	*P.L. Class IsAbstract*	*P.L. Class IsEnum*	*Implementation ___ support*	*___ support alias*	*Comments*
						SHOULD try to resolve the selection without the alias.
Instantiable class	NO	NO	NO	MUST	MUST NOT	If an alias is used, the implementation MUST throw an error.
Enumeration	NO	NO	YES	SHOULD	MUST	Implementations that do not support enumeration types MUST return an error. The individual enumerator MUST be specified as an alias. Eg. `getService(get-class(Example.EventHandlerEnum), 'LogicHandler')`

If the selectors cannot be resolved into an instance, an error of type `Progress.Lang.AppError` MUST be thrown by the implementation. A null object MUST NOT be returned.

LIFECYCLE MANAGEMENT

The Service Manager is responsible for the management of the implementing objects' lifecycles, including startup, caching and managed destruction. Objects created and returned by the `getService` API always have an associated lifecycle scope. This scope indicates the expected lifetime of the object, and includes values like "session" (effectively a singleton), "request" and "container", which is a container of the application's choosing.

Scopes have a type (an OO interface name) and a value (returned by a `getScope` API).

Scope type	*Meaning*	*Destruction*	*Scope value*
Transient	The lifecycle of the instance is not managed by the Service Manager. One instance, running free, like a wolf. Practically, this means a new instance of the type will be created each time one is	Standard/normal garbage-collection rules apply	This MUST be the unknown / null value

Scope type	*Meaning*	*Destruction*	*Scope value*
	requested. This is typically the default scope if none is specified.		
Session	Only a single instance of the type will be created in an ABL session, and the same instance will be returned for each subsequent request.	Will be destroyed when the AVM session is terminated	PASOE sessions have an identifier accessible from the `session:current-request-info:SessionId` property ; client and Classic AppServer sessions do not. Implementers SHOULD use a non-blank value.
Request	One instance of the type will be created for each request made to an AppServer. Does not apply to interactive clients.	Destroyed after a request, typically as part of an AppServer's `deactivate` event procedure. Requires an explicit `stopServices` call.	AppServer sessions have an identifier accessible from the `session:current-request-info:RequestId` property
Container	Custom scope, as determined by an implementer. Can be thought of as a "named scope".	Requires an explicit `stopServices` call made by the application/framework.	This value is entirely application-dependent. It SHOULD NOT be the unknown/null value

Scope types are typically provided as part of the configuration data. Scope values are needed for some scope types - "container" in particular.

The Service Manager provides a `stopServices` API which takes a scope indicator as a parameter. This API will destroy and remove from its cache(s) all instances that were started with the specified scope. Generally-speaking , this API is intended to be called when a request, session or some other application-defined context is complete.

Objects that implement the `IService` interface MUST have their `dispose` method called before deletion.

Calling the `stopServices` API for transient scope is undefined behaviour and SHOULD NOT return an error.

LIFECYCLE SCOPES

Scopes will be defined as a set of interfaces that MUST ultimately inherit from a `Ccs.ServiceManager.ILifecycleScope` type. This is to allow a Service Manager to gracefully follow a path if a particular scope is not supported by the implementation. For example,

`Example.A.ServiceManager` may only support Session Scope but MUST gracefully handle a `stopServices` call which has a Request Scope argument passed in.

A scope MAY return some data to identify the scope via the `getScope` method, which returns specifics of the Scope. For instance, for a Request scope it would return the value of the `session:current-request-info:RequestId`.

This specification will define the initial hierarchy of scopes. Implementers MUST support Request and Session scope to be spec-compliant.

CONFIGURATION DATA

This specification does NOT describe to format of the configuration data used by the Service Manager, nor its location nor means of loading.

This specification RECOMMENDS that the storage of configuration data be kept separate from the Service Manager implementation (ie don't add a CASE statement into the `getService` API).

Configuration data MUST at a minimum contain - a primary selector - an implementing type

The data SHOULD contain a lifecycle scope, and MAY contain any additional information to support the functionality of the Service Manager (eg for Dependency Injection purposes).

Component Architecture

Component Package Definition

The Service interface will be in the `Ccs.Common` package. The Service Manager and other components (scopes) will be in the `Ccs.ServiceManager` package.

Component Property Data and Organization

In terms of the naming conventions, storage organization and interaction between components, the requirements found in the document CCS Specification: OpenEdge Application Architecture Version 1 also apply to the components defined in this specification.

Component Run-time Characteristics

The Service Manager will be instantiated by the session bootstrap mechanism, which SHOULD be an implementation of the `Ccs.Common.IStartupManager`.

The Service Manager SHOULD be run as a singleton to allow for objects to be cached within its scope (ie if the Service Manager is destroyed, it is reasonable to assume that any caches it

maintains are flushed/cleared). The lifecycle of the Service Manager itself SHOULD be at least as long as the longest / broadest lifecycle scope it supports.

It is RECOMMENDED that the Service Manager be one of (if not) the first components started in an application and one of (if not) the last components destroyed in the application.

Since the Service Manager implements the `IManager` and `IService` interfaces, the `initialize()` method MUST be run at startup.

The Service Manager MAY choose to load configuration data during the initialisation phase. If data is loaded during startup, it is RECOMMENDED that this be done in the `initialize()` method and not a constructor.

Recognising that global state is not considered a best practice, it is RECOMMENDED that the Service Manager instance be globally available to an application. This is to avoid having to make sweeping changes to all application objects/services to accept a reference the Service Manager, or to traverse the tree of extant objects whenever a new service is requested.

If an application implements the OEAA (as defined by the CCS Architecture Specification) then, once instantiated, the reference to the Service Manager MUST be set on the `Ccs.Common.Application`'s static `ServiceManager` property. This is REQUIRED in order to comply with the CCS Architecture Specification.

If an application does not desire compliance to the CCS Architecture Specification, then it SHOULD be set on the `Ccs.Common.Application`'s static `ServiceManager` property.

At this point the Service Manager MUST be ready to accept requests.

The Service Manager SHOULD delegate requests for services that are Managers to the Startup Manager, which is accessible from the `Ccs.Common.Application`'s static `StartupManager` property. The `getManager` API is provided for this purpose. Implementers should take care to ensure that an infinite loop does not result where the Startup Manager calls the Service Manager which calls the Startup Manager ad inifinitum for the same manager type.

Implementers should take care to ensure that a request for the ServiceManager returns itself.

Component Error Handling

It is preferred that a fail fast philosophy be followed during initialization. Errors SHOULD be left to bubble up through the call stack to the session startup or service interface routines rather than being intercepted.

Errors MUST have messages, and MUST NOT rely on a `Progress.Lang.AppError`'s `ReturnValue` property for message strings. Thus when the default constructor or the constructor

taking only a string is used to instantiate an `Progress.Lang.AppError`, it MUST be followed with a call to the `AddMessage()` method before the error is thrown.

An implementation MAY catch errors thrown and return them with a code as described below.

ERROR CODES

The error codes and texts are suggested values for errors thrown by an implementation.

Error type	*Code*	*Text*	*Usage*
Progress.Lang.AppError	2000	Unhandled error: <error-message>	
Progress.Lang.AppError	2001	Service implementation cannot be found for <service-name >	
Progress.Lang.AppError	2002	Unsupported service type: <service-type: abstract class, class, interface, enumeration>	
Progress.Lang.AppError	2003	Invalid <argument-name: alias, scope, service name> argument <argument-value>	
Progress.Lang.AppError	2004	Invalid request for service type <service-type: abstract class, class, interface, enumeration> with argument <argument-name: alias, scope>	

Dependencies and interactions with other common standards

Component	*Nature*
Ccs.Common.IStartupManager	A request for an instance of IManager MUST be delegated to the Startup Manager (if one is stored in the `Ccs.Common.Application`'s `StartupManager` property)
Ccs.Common.Application:ServiceManager	A global reference to the Service Manager. Serves two purposes: to keep the Service Manager alive; to provide access from anywhere in a session without passing a reference

3. COMPONENT INTERACTION WITH EXTERNAL SUB-SYSTEMS

As stated in the Introduction, the Service Manager's purpose is to provide references to various application components. The Service Manager may thus be used (called) by a wide variety of components.

4. COMPONENT INTERFACES AND CLASSES

All method parameters are INPUT unless otherwise noted by an IN-OUT or OUT prefix.

Ccs.Common.IService

An OPTIONAL common base interface for all application services. REQUIRED for OEAA Manager types (via the `Ccs.Common.IManager` interface).

An implementing class SHOULD have a default (no-argument) constructor.

Ancestors

None.

Methods

Name	*Return type*	*Parameter list*	*Description*
initialize	void	none	Initializer/Startup
dispose	void	none	Shutdown/Anti-Initializer.

Properties

There are no properties specified for this type.

Events

There are no events specified for this type.

Ccs.Common.IServiceManager

The `IServiceManager` interface represents the public API of the component. All interactions with the Service Manager component MUST be done through this interface.

Ancestors

`Ccs.Common.IManager` which inherits from`Ccs.Common.IService`

Methods

In addition to the methods specified by the ancestor types,

Name	*Return type*	*Parameter list*	*Description*

getService	Progress.Lang.Object	service-name (Progress.Lang.Class)	Returns a usable instance of the requested service.
getService	Progress.Lang.Object	service-name (Progress.Lang.Class), scope (Ccs.ServiceManager.ILifecycleScope)	Returns a usable instance of the requested service
getService	Progress.Lang.Object	service-name (Progress.Lang.Class), alias (character)	Returns a usable instance of the requested service
stopServices	void	scope (Ccs.ServiceManager.ILifecycleScope)	Destroys and flushes from any cache(s) objects scoped to the argument scope.

Properties

There are no additional properties specified in addition to those specified by the ancestor types.

Events

There are no additional events specified in addition to those specified by the ancestor types.

Ccs.ServiceManager.ILifecycleScope

Ancestors

None

Methods

Name	***Return type***	***Parameter list***	***Description***
getScope	Progress.Lang.Object	none	Returns a unique identifier for this scope. See the Component Description for example values

Properties

There are no properties specified for this type.

Events

There are no events specified for this type.

Ccs.ServiceManager.ITransientScope

A scope that indicates that the lifecycle of the instance is NOT controled/managed by the Service Manager. This is a marker interface (it does not extend its ancestors' members).

Ancestors

`Ccs.ServiceManager.ILifecycleScope`

Methods

Name	*Return type*	*Parameter list*	*Description*
getScope	Progress.Lang.Object	none	Returns a unique identifier for this scope. For transient scopes this value MUST be an the unknown value (not-valid object)

Properties

There are no additional properties specified in addition to those specified by the ancestor types.

Events

There are no additional events specified in addition to those specified by the ancestor types.

Ccs.ServiceManager.ISessionScope

A scope that indicates an entire ABL/AVM session. This is a marker interface (it does not extend its ancestors' members).

Ancestors

`Ccs.ServiceManager.ILifecycleScope`

Properties

There are no additional properties specified in addition to those specified by the ancestor types.

Events

There are no additional events specified in addition to those specified by the ancestor types.

Ccs.ServiceManager.IRequestScope

A scope that indicates a single requrest to an Application Server. This is a marker interface (it does not extend its ancestors' members).

Ancestors

`Ccs.ServiceManager.ISessionScope`

Properties

There are no additional properties specified in addition to those specified by the ancestor types.

Events

There are no additional events specified in addition to those specified by the ancestor types.

Ccs.ServiceManager.IContainerScope

A implementation-defined scope. This is a marker interface (it does not extend its ancestors' members).

Ancestors

`Ccs.ServiceManager.ISessionScope`

Properties

There are no additional properties specified in addition to those specified by the ancestor types.

Events

There are no additional events specified in addition to those specified by the ancestor types.

5. ASYNCHRONOUS APPLICATION CALLBACKS

6. COMPONENT DESIGN AND IMPLEMENTATION

ABL source code for the Service Manager is also available on GitHub *(URL tbd)*. This published specification provides the canonical definition of the Service Manager; in cases where the spec and code differ, this document wins.

Copyright notice

For brevity, copyright headers have been removed from the code below. The standard CCS copyright header is

```
/*------------------------------------------------------------------------
  This Software is licensed by Progress Software Corporation (licensor)
  under the Progress Software Common Component Specification Project
  Release License Agreement available at
  https://community.progress.com/products/directions/common_component/p/releaselicenseagreement

  The Interface definition is part of the Common Component Specification [CCSSVCMGR01]. The
  file is considered as a Specification Implementation Condition as described
  in section 2.1.1.1: If Licensor has made Specification Implementation
  Conditions available as of the date Licensee completes its Independent
  Implementation, then Licensee must, prior to making any claim that its
  Independent Implementation complies with the Specification, ensure that
  the Independent Implementation satisfies all of the Specification
  Implementation Conditions. If Licensor subsequently makes available or
  updates, from time to time, the Specification Implementation Conditions,
  then Licensee will verify that its Independent Implementation satisfies the
  latest version of the Specification Implementation Conditions within ninety
  (90) days following Licensor's release thereof.

  Contributors:
    <name>, <organisation> <date>
----------------------------------------------------------------------*/
```

Ccs.Common.IService

```
/*------------------------------------------------------------------
   File       : IService
   Purpose    : Base service definition
   Author(s)  : pjudge@progress.com
   Created    : 2016-09-26
   Notes      :
 ----------------------------------------------------------------*/
interface Ccs.Common.IService:

   /*  Initializer/Startup */
   method public void initialize().

   /* Shutdown/Anti-Initializer */
   method public void dispose().

end interface.
```

Ccs.Common.IManager

The definition of the `IManager` interface is taken from the Startup Manager Specification *(Prinsloo et al, 2016)* and is included here for completeness

```
USING Ccs.Common.IService FROM PROPATH.

INTERFACE Ccs.Common.IManager INHERITS IService:

END INTERFACE.
```

Ccs.Common.IServiceManager

```
/*------------------------------------------------------------------
   File       : IServiceManager
   Purpose    : Base Service Manager interface
   Author(s)  : pjudge@progress.com
   Created    : 2016-09-26
   Notes      :
```

```
 ------------------------------------------------------------------------*/

interface Ccs.Common.IServiceManager inherits Ccs.Common.IManager:
  /* Returns a usable instance of the requested service.

    @param P.L.Class The service name requested
    @return P.L.Object A usable instance
    @throws P.L.AppError Thrown when no implementation can be found */
  method public Progress.Lang.Object getService(
                        input poService as class Progress.Lang.Class).

  /* Returns a usable instance of the requested service.

    @param P.L.Class The service name requested
    @param ILifecycleScope A requested scope. The implementation may choose to
                      ignore this value.
    @return P.L.Object A usable instance
    @throws P.L.AppError Thrown when no implementation can be found */
  method public Progress.Lang.Object getService(
                    input poService as class Progress.Lang.Class,
                    input poScope as Ccs.ServiceManager.ILifecycleScope).

  /* Returns a usable instance of the requested service.

    @param P.L.Class The service name requested
    @param character An alias for the service. The implementation may choose
                to ignore this value.
    @return P.L.Object A usable instance
    @throws P.L.AppError Thrown when no implementation can be found */
  method public Progress.Lang.Object getService(
                        input poService as class Progress.Lang.Class,
                        input pcAlias as character).

  /* Destroys and flushes from any cache(s) objects scoped to the argument
    scope.

    @param ILifecycleScope A requested scope for which to stop services. */
  method public void stopServices(
                    input poScope as Ccs.ServiceManager.ILifecycleScope).

end interface.
```

Ccs.ServiceManager.ILifecycleScope

```
/*------------------------------------------------------------------------
   File       : ILifecycleScope
```

```
   Purpose     : Marker for lifecycle scopes
   Author(s)   : pjudge@progress.com
   Created     : 2016-09-26
   Notes       :
 ----------------------------------------------------------------------*/
interface Ccs.ServiceManager.ILifecycleScope:

   /* Returns a unique identifier for this scope.

      @returnProgress.Lang.Object An identifier for the scope. May be a wrapper
            around an ABL primitive. May be unknown. */
   method public Progress.Lang.Object getScope().

end interface.
```

Ccs.ServiceManager.ITransientScope

```
/*----------------------------------------------------------------------
   File       : ITransientScope
   Purpose     : Transient scope definition - lifecycle is not managed by
                 the Service Manager
   Author(s)   : pjudge@progress.com
   Created     : 2016-09-26
   Notes       :
 ----------------------------------------------------------------------*/

interface Ccs.ServiceManager.ITransientScope inherits Ccs.ServiceManager.ILifecycleScope:

end interface.
```

Ccs.ServiceManager.ISessionScope

```
/*----------------------------------------------------------------------
   File       : ISessionScope
   Purpose     : Will be destroyed when the AVM session is terminated
   Author(s)   : pjudge@progress.com
   Created     : 2016-09-26
   Notes       :
 ----------------------------------------------------------------------*/

interface Ccs.ServiceManager.ISessionScope inherits Ccs.ServiceManager.ILifecycleScope:
```

```
end interface.
```

Ccs.ServiceManager.IRequestScope

```
/*------------------------------------------------------------------------
   File       : IRequestScope
   Purpose    : Destroyed after a request, typically as part of an AppServer's
                deactivate event procedure. Requires an explicit stopServices call.
   Author(s)  : pjudge@progress.com
   Created    : 2016-09-26
   Notes      :
 ----------------------------------------------------------------------*/

interface Ccs.ServiceManager.IRequestScope inherits Ccs.ServiceManager.ILifecycleScope:

end interface.
```

Ccs.ServiceManager.IContainerScope

```
/*------------------------------------------------------------------------
   File       : IContainerScope
   Purpose    : Custom application lifecycle scope
   Author(s)  : pjudge@progress.com
   Created    : 2016-09-26
   Notes      :
 ----------------------------------------------------------------------*/

interface Ccs.ServiceManager.IContainerScope inherits Ccs.ServiceManager.ILifecycleScope:

end interface.
```

Design method and standards

Naming conventions

Programming standards

7. REFERENCES

Chase, S. B., Elwell, R., & Smith, R. (2016). *CCS Specification: OpenEdge Application Architecture Version 1*. Progress Software.

Ormerod, M. (2006). *Defining The Openedge® Reference Architecture - Common Infrastructure.* Progress Software.

Prinsloo S. L., Elwell, R., Judge, P., & Smith, R. (2016). *Startup Manager Specification OpenEdge Application Architecture Specification (OEAA) version 1.0.* Progress Software.

8. DOCUMENT CONTROL

Document attribute	***Value***
Title	Service Manager Specification OpenEdge Application Architecture Specification (OEAA)
Version	1.0.0
Last-Update-At	2016-09-26
Last-Update-By	@PeterJudge-PSC

SPECIFICATION TEAM

Name	***Email / GitHub***	***Organisation***
Shelley B. Chase	schase@progress.com / @sbschase	Progress Software Corp
*Marian Edu**	marian.edu@acorn.ro / @akera-io	Acorn IT
Rom Elwell	rome@issol.com / @romelwell	Innovative Software Solutions
*Mike Fechner**	mike.fechner@consultingwerk.de / @mikefechner	Consultingwerk
Ganesh Iyer+	gaiyer@progress.com / @ganeshn9	Progress Software Corp
*Peter Judge**	pjudge@progress.com / @PeterJudge-PSC	Progress Software Corp
Christopher Longo	chlongo@progress.com	Progress Software Corp
Paul Moberg	paulmoberg@quickenloans.com	Quicken Loans
Mark Opfer	mopfer@dmsi.com / @MarkOpferDMSI	DMSi
*Simon L. Prinsloo**	simon@vidisolve.com / @SimonLPrinsloo	Vidisolve
*Robin Smith**	rosmith@progress.com / @RobinSmith-PSC	Progress Software Corp

* contributing author
+ team lead

9. DOCUMENT HISTORY

Date	*Author*	*Comment*
2016-09-20	@PeterJudge-PSC	Initial draft for internal review
2016-09-21	@PeterJudge-PSC	Updated with feedback from initial internal review mtg
2016-09-26	@PeterJudge-PSC	Updated after email review. Feedback from Marian Edu, Mike Fechner
2016-12-19	@PeterJudge-PSC	Updated after community review. Feedback from Carl Verbiest

10. OUTSTANDING ISSUES