**Document Identifier: DSP2000**

**Date: 2015-04-27**

**Version: 2.0.0**

# CIM Diagnostic Model White Paper
# CIM Version 2.34

**Supersedes: 1.0.0**

**Document Type: White Paper**

**Document Class: Informative**

**Document Status: Published**

**Document Language: en-US**

# CONTENTS

# Figures

# Abstract

Diagnostics is a critical component of systems management. Diagnostic services are used in problem containment to maintain availability, achieve fault isolation for system recovery, establish system integrity during boot, increase system reliability, and perform routine preventive maintenance. The goal of the Common Diagnostic Model (CDM) is to define industry-standard building blocks based on, and consistent with, the DMTF Common Information Model (CIM) that enable seamless integration of vendor-supplied diagnostic services into system and SAN management frameworks.

In this paper, the motivation behind the CDM is presented. In addition, the core architecture of the CDM is presented in the form of a diagnostic schema. Proper usage of the schema extensions is presented in a tutorial manner. Future direction for the CDM is discussed to further illustrate the motivations driving CDM development, including interoperability, self-management, and self-healing of computer resources.

# Introduction

The **Common Diagnostic Model (CDM)** is both an architecture and methodology for exposing system diagnostic instrumentation through standard CIM interfaces. The schema has been extended to improve versatility and extendibility. A number of major changes occurred since the previous version of this white paper.

The purpose of this paper is to describe the CDM schema as it appears in CIM 2.34 and describe future development. This paper provides guidance, where appropriate, to client and provider implementers to reinforce the standardization goal. Guidance for diagnostic test developers is not within the scope of this whitepaper and is being documented by the CDM Forum.

# CIM Diagnostic Model White Paper CIM Version 2.34

## 1 Executive summary

This paper explains how CDM standardizes diagnostics into a generic management framework that enhances health and fault management. Adopters can implement any of a number of available concrete profiles that can be extended as needed by the vendor. For components that do not have an existing concrete profile, adopters can simply base their implementation on the documented diagnostic design pattern.

The current versions of the model are first presented and described in detailed followed by areas of future development.

### 1.1 Overview

The term **diagnostics** has been used to describe a variety of problem-determination and prevention tools, including exercisers, excitation/response tests, information gatherers, configuration tools, and predictive failure techniques. This paper adopts a general interpretation of this term and addresses all forms of diagnostic tools that would be used in OS-present and preboot environments. This paper addresses general enabling infrastructure and specific diagnostics are deferred to specific diagnostic profiles.

The OS-present environment presents a formidable set of challenges to diagnostics programmers. They must deal with system status and information hidden behind proprietary APIs and undocumented incantations. Although CIM remedies this situation, diagnostics programmers are also faced with OS barriers between user space and the target of their efforts, making it difficult, often impossible, to manipulate the hardware directly. The CDM eases this situation through a standardized approach to diagnostics that uses the more sophisticated aspects of CIM—the ability to manipulate manageable system components by invoking methods.

### 1.2 Goals

The goals of the CDM are:

- Manageability through standardization
- Interoperability
- Diagnostic effectiveness
- Global access
- Life cycle applicability
- Enable health and fault management
- Integration with other management functions
- Integration with other management initiatives
- Extendable to other system elements

### 1.2.1 Manageability through standardization

Faced with the requirement to deliver diagnostic tools to their customers, chip and adapter developers have to deal with a variety of proprietary APIs, report formats, and deployment scenarios. The CDM specifies a common methodology, with CIM at its core, which results in a "one size fits all" diagnostic package. Diagnostic management applications can obtain information about which diagnostic services are available, configure and invoke diagnostics, monitor diagnostic progress, control diagnostic execution, and query CIM for information that the diagnostic service gathers.

If the CDM methodology is followed, these standard diagnostic packages can be incorporated seamlessly into applications that are implemented as CIM clients. The diagnostic programmer, relieved from the effort associated with satisfying multiple interfaces, can spend more time improving the effectiveness of the tools.

Standardization also has allowed the creation of a number of both client and server libraries for supporting the interface. For example, JSR48 provides a Java library for interfacing between clients and servers (see JSR48). In addition, there are common tools available for debugging code developed to the standard.

The CMPI (Common Manageability Programming Interface) defines a common C-based provider interface (see CMPI). With this definition, a provider can be re-used in any management server environment supporting this interface.

### 1.2.2 Interoperability

Diagnostic CIM models extend the CIM models to address diagnostic capabilities. The CIM interface between CIM clients (CIM client libraries and applications) and WBEM servers (object managers and providers) is standardized and is platform-neutral. The implementations of CIM clients and providers do not have to be platform-neutral. A single provider implementation can support multiple clients and a single client can talk to multiple providers using a single standard interface. To the extent that CIM implementations promote interoperability, so does the CDM. These CDM implementations allow clients to manage diagnostic assets across heterogeneous platforms and environments.

### 1.2.3 Diagnostic effectiveness

Behind the CDM infrastructure are the diagnostic tools themselves. When developed to the CDM, the tools become less difficult to deploy and the effectiveness of the entire package can be improved. Several factors are at play. Ease of deployment through standardization and interoperability increases availability, thus expanding coverage. Tool developers also have the entire CIM model implementation for other aspects of device management to draw on in their problem-determination and resolution efforts. By integrating diagnostics with other aspects of device management (e.g., configuration management or performance monitoring), the CDM also goes beyond base diagnostic features by recommending techniques to vendors that lead to integration of diagnostics into device drivers, thus gaining access to more details of the device being diagnosed. The effectiveness of the diagnostics is improved by integrating with all of the available system information.

Being able to bind diagnostics to the same elements that you are targeting for other management operations is not only extremely powerful but invaluable. CDM makes this possible by standardizing diagnostics on a well known and established management framework.

Diagnostics are fine tuned, not just to the component, but also to its environment enabling a more comprehensive view and control of component status and health. Diagnostics are consequently executed in a truly holistic manner such that critical business services and workflows are not adversely impacted. Workloads receive the resources they need and when they need them with an understanding of what elements will be affected. Having intelligent control of individual element states makes it possible to achieve an overall desired state for the environment.

CDM unifies diagnostics into one consolidated system that permits managing different resource types such as Network, Storage, and Compute. Resource types are normalized across vendors so that diagnostic information can be consumed in a consistent manner. This helps free CDM adopters from many of the infrastructure issues and allows them to focus almost exclusively on diagnostic content.

### 1.2.4 Global access

The CIM framework is designed for managing system elements across distributed environments and can support these elements without regard to locale. This feature greatly expands the scope in which it can be deployed and utilized without special adaptations or additional costs. This facilitates cost-effective serviceability scenarios and warranty-expense reduction.

### 1.2.5 Enhances ITIL processes

CDM provides a standard way for ITIL processes to support problem verification and isolation.

### 1.2.6 Life cycle applicability

The CDM is designed to be applicable through and in all stages of a product’s life cycle. For example, the same set of tests that was used during design and development can also be executed to verify a component on the manufacturing floor before it is shipped and later in a customer’s production environment. The earlier in the life cycle that errors are detected the cheaper they are to fix. And the more errors that are caught before a component gets into a customer’s hands, the more satisfied the customer.

### 1.2.7 Enable health and fault management

Diagnostics is an integral part of health and fault management of a system or device. When diagnostics are combined with basic management functions of monitoring and configuration of systems and devices the result is a robust environment for health and fault management. The combination of management profiles for systems and devices and CDM interfaces for diagnostics enables verifying the health of systems, determining what elements are impacted by a failing element, doing failure prediction and repairing or reconfiguring failed devices.

### 1.2.8 Integration with other management functions

Integrating diagnostics interfaces with other management functions allows clients to access other elements impacted by the failing elements. For example, a failing hard disk drive (HDD) impacts higher level management elements (e.g., Storage Volumes) that store data on the HDD. CDM puts diagnostics information in the context of other management functions modeled in CIM.

The CDM design includes linkage to “affected elements” of the tested components. Repair actions may require actions on the affected elements, as well as the tested component. By being integrated into the CIM management model, functions required to reconfigure or repair the affected elements are discovered and readily available.

### 1.2.9 Integration with other management initiatives

Initial work on CDM has focused on diagnostics for physical elements of a system. However the CDM concepts can be applied to any element of a system. This allows CDM to be integrated with other CIM based management profiles (like networks and external devices) and initiatives (like cloud computing, server management, or storage networking).

For example, the Storage Networking Industry Association (SNIA) is using CDM and its diagnostics capability to enhance its management functions for health and fault management. By integrating the basic diagnostics of the CDM model with its existing storage management profiles, SNIA will be providing a robust system for the health and fault management of storage environments and specifically storage devices.

### 1.2.10 Extendable to other system elements

CDM defines an abstract profile containing general constructs for implementing diagnostic tests, controlling test execution, and monitoring results. This abstract profile can be applied to any managed element in a system.

CDM also has a set of "concrete" profiles for managing specific elements in a system (e.g., CPU, HDDs, Host Bus Adapters). These concrete profiles identify specific tests and results for the specific devices that are supported. However, it is important to note that similar concrete profiles can be created for any other managed elements in a system. Those elements do not have to be "physical" elements. They can be logical elements such as filesystems or logical volumes. Regardless of whether they are physical or logical, elements diagnostics for the target element are made available in exactly the same way.

## 1.3 Who should read this paper

This paper was prepared to help developers (of diagnostics and system management in general) understand the CIM components of the Common Diagnostic Model and other areas of the model that fulfill the requirements of a comprehensive health and fault management methodology for modern computer systems. This paper may also be used by system professionals that want to understand how diagnostics fit in the overall management of systems. Anyone planning to use or create diagnostic services should read it.

This paper assumes some basic knowledge of the CIM Schema, represented by the MOF files. Detailed information in these files will not be covered in this paper.

This paper deals primarily with the CDM architecture. The CDM also includes implementation standards to promote OEM/vendor interoperability and code reuse. The reader can refer to specific CDM profiles for implementation details (see the Bibliography for the list of current profiles). This document also addresses issues related to compliance. Tools are being developed to validate CDM compliance to assist in validation of tools and tests that claim support of CDM.

## 1.4 CDM versions

CDM version 1.0 (CDMV1) was introduced in CIM 2.3. It has been enhanced in subsequent versions of the CIM Schema. Some of the model components peculiar to CDMV1 have been deprecated prior to the introduction of CDM version 2.0, at which time support for CDMV1 clients and providers has been discontinued.

CDM version 2.0 (CDMV2) was introduced with CIM Schema 2.9 and has evolved to CDM version 2.1 (CDMV2.1) and CIM Schema 2.34. The settings/test/results concept is still present, but it is modeled using services, jobs, and logs. In addition, CDM version 2.1 has introduced support for interactive tests and alert indications as a means of reporting test events as standard messages to clients.

## 1.5 Conventions used in this document

Classes and properties are written using capitalized words without spaces, as in ManagedElement (contrast with "managed element," which is the generic form).

The **Bold** attribute is added for visual impact with no other implied meaning.

Methods include parentheses ( ) for quick identification, as in RunDiagnosticService( ).

Arrays include brackets [ ] for identification, as in LoopControl[ ].

A colon between class names is interpreted as “derived from,” as in ConcreteJob : Job.

A “dot” between a class name and a property name is interpreted as “containing the property,” as in Capabilities.InstanceID. (InstanceID is a property of the Capabilities class.)

The prefix “CIM_” is often omitted from class names for brevity and readability.

# 2 Terms and definitions

The following terms are used in this document:

**2.1**
**Diagnostic Job**
Thread for executing a diagnostic service (such as a Diagnostic test)

**2.2**
**Interactive Test**
Test that solicits input from a client application to be completed

# 3 Symbols and abbreviated terms

The following abbreviations are used in this document:

**3.1**
**CDM**
Common Diagnostic Model

**3.2**
**CDMV1**
Version 1 of the CDM (based on CIM 2.3)

**3.3**
**CDMV2**
Version 2.0 of the CDM (based on CIM 2.9)

**3.4**
**CIM**
Common Information Model

**3.5**
**CR**
(CIM) Change Request

**3.6**
**DBCS**
Double Byte Character Set

**3.7**
**FRU**
Field Replaceable Unit

**3.8**
**ME**
ManagedElement

**3.9**
**MOF**
Managed Object Format

**3.10**
**MSE**
ManagedSystemElement (the class or its children)

**3.11**
**NLS**
National Language Support

**3.12**
**RAS**
Reliability, Availability, and Serviceability

**3.13**
**SAN**
Storage Area Network

**3.14**
**UML**
Unified Modeling Language

**3.15**
**WBEM**
Web Based Enterprise Management

**3.16**
**XML**
Extensible Markup Language

# 4 Modeling diagnostics

The Common Diagnostic Model (CDM) extends the CIM Schema to cover the management of diagnostics, including diagnostic tests, executives, monitoring agents, and analysis tools. The objective of diagnostic integration into CIM is to provide a framework in which industry-standard building blocks that contribute to the ability to diagnose and predict the system’s health can seamlessly integrate into enterprise management applications and policies. This clause discusses the modeling concepts that are relevant to implementing diagnostics with CIM.

## 4.1 Consumer-provider protocol

A CIM diagnostic solution has two components: diagnostic consumers (or diagnostic CIM clients) and diagnostic providers. Diagnostic providers register the classes, properties, methods, and indications that they support with the CIM object manager (CIMOM). When a management client queries CIM for diagnostics supported on a given managed element, CIM returns the instances of the diagnostic services associated with that managed element. This action establishes communication between the discovered diagnostic providers and the management client. The management client can now query CIM for properties, enable indications, or execute methods according to the standard and the diagnostic protocol conventions described in this document. The conventions that diagnostic consumers and providers must follow the rules and behavior defined in the profiles defined by CDM.

## 4.2 Implementation-neutral interface

The diagnostic interface is implementation neutral. Implementations present their functions and data through a standard interface that is independent of how the functions are implemented or how the data is actually represented in the system or device.

Implementations of the interface may be:

- Re-entrant or not
- Single threaded or multithreaded
- Dynamically loaded or “always resident”
- Implemented with any number of providers
- Resident on the system or remote
- The execution environment the provider uses
- The language the provider is written in

## 4.3 Backward compatibility

CDM version 2.1 is backward compatible with CDM version 2.0. That is, elements of the model (and interface) that were supported in version 2.0 are supported in version 2.1. Version 2.1 adds elements and functions that were not present in version 2.0. However, a client that was written to version 2.0 should work with a version 2.1 implementation of CDM. The client would not do anything with the new elements or functions introduced in version 2.1 and all the elements and functions of 2.0 would be present and should work as they did in version 2.0.

In addition, to the extent that implementations of CDM version 2.0 conform to the abstract Diagnostics Profile (DSP1002 version 2.0), implementations of concrete profiles (e.g., the FC HBA Diagnostics profile) should be backward compatible with the elements in DSP1002 version 2.0. This is because the concrete profiles are based on CDM version 2.0 or CDM version 2.1. While certain tests may not be present, many tests (e.g., Ping and Echo), if implemented to 2.0 should be compatible with implementations of 2.1 of the concrete profile. This is because the base function (RunDiagnosticService) has not changed.

## 4.4 Extendable to other Diagnostic Services for health and fault management

Diagnostics are more than just test applications. The goal is to make CDM extendable to other diagnostics related capabilities. Overall diagnostics create controlled stimuli and monitor, gather, record, and analyze information about detected faults, state, status, performance, and configuration. Because of its diverse uses, diagnostics are best modeled as a service that launches or enables the components necessary to implement the diagnostic actions requested by the client.

These diagnostic components may be implemented as test applications, monitoring daemons, enablers for built-in diagnostic capabilities, or proxies to some other instrumentation that is implemented outside of CIM.

## 4.5 Diagnostics are applied to managed elements

Diagnostics are applied to managed elements. "Applied" means that a test checks a managed element, a diagnostic daemon monitors a managed element, diagnostic instrumentation is built into the managed element, and so on. One of the goals of CIM-based diagnostics is the packaging of diagnostics with the vendor's deliverable or Field Replaceable Unit (FRU). Thus diagnostics are often applied at the FRU level of granularity.

Diagnostic services are commonly applied to:

- **Logical Devices**: Most vendor-supplied diagnostics are for add-on peripherals such as adapters and storage media. In this case clear correspondence exists between the diagnostic's scope and a CIM-defined logical device class.
- **Systems**: Not all diagnostic use cases have coverage that corresponds to logical devices. Some diagnostic services are best applied to a system as a single functional unit that is scoped to it as a FRU. Some examples are:
  - System stress tests and monitors that measure aggregate system health
  - Miscellaneous, non-modeled, or baseboard devices that are often best viewed as part of a system-level FRU
  - Controllers that are part of an internal system bus structure and may not be independently diagnosable but must be tested by proxy through another logical device
    In this case, the controller is an embedded, indistinguishable component that contributes to the overall system health.
- **Other Services**: Diagnostic services may also be applied to other non-diagnostic services. These diagnostics may be used to ensure the reliability of the associated service.

## 4.6 Generic framework

Diagnostic services share the semantics of the CIM model regardless of whether the service launches tests, starts a monitoring agent, or enables instrumentation. They share the same mechanisms for publishing, method execution, parameter passing, message logging, and reporting FRU information.

By integrating the diagnostic model into the other areas of the CIM model, the client application can easily transition between the management model and the diagnostics for the elements managed. Examples include the "jobs" model for monitoring, the "log" model for capturing information, indications for reporting test results, and effective use of the logical and physical models.

### 4.6.1 Diagnostic control

Diagnostic clients may need to control and monitor the status and progress of the diagnostics elements that the service provider launches to implement a service request. Clients achieve this control and monitoring capability in a generic manner by using the CIM job and process model. The Diagnostics profile uses an extended version of the DMTF Job Control Profile to do this. The diagnostics extensions for job control are backward compatible with the DMTF Job Control profile. That is, they extend, but do not change the basic elements of the profile. The elements launched by the diagnostic service can be collectively controlled and monitored through an instance of ConcreteJob that is returned by the diagnostics RunDiagnosticService method in the diagnostic service.

### 4.6.2 Diagnostic logging and reporting assumptions

Diagnostics require the ability to report information about detected faults, state of the device, and performance on the device. Diagnostics must also report the status of the diagnostics service and configuration of the diagnostic components. This information can be gathered dynamically at checkpoints while the diagnostic service is active (for concurrent analysis) or after the service is complete (for postmortem analysis). Diagnostics use alert indications and a log to record relevant information from diagnostic service applications, agents, and instrumentation.

The diagnostic model also uses other CIM models for standardizing error codes and indications. The error codes and indications may be used to create trouble tickets and integrate CIM diagnostics into CIM-based industry standard diagnostic policies and RAS use cases.

### 4.6.3 Localization

Localization refers to the support of various geographical, political, or cultural region preferences, or locales. A client may be in a different country from the system it is querying and would prefer to be able to communicate with the system using its own locale. Inherent differences, such as language, phraseology, and currency, must be considered.

CDM communicates to clients using standard messages. These are messages that include text and "substitution variables". The text may be translated. For example, CDM uses standard messages to communicate errors or warnings. One specific example would be the message DIAG4:

> The <Diagnostic Test Name> test on the selected element to test <Element Moniker> completed with warnings. See earlier warning alert indications or the <Log Object Path> for more details.

The substitution variables are denoted by the angle brackets (<variable>). The rest of the message is just text that may be localized. The substitution variables are taken from the model instances (e.g., <Log Object Path>) and should not be translated.

# 5 CDMV2.1

CDM provides a robust structure for discovering diagnostic tests, running and monitoring them, and reporting results. CDMV2.1 supports a flexible and extendable model based on settings/services/jobs/logs.

The following diagram represents the model components unique to CDM. You can find related components (for example, disk drive) by searching the online documentation at www.dmtf.org.

This document corresponds to CIM 2.34 and DSP1002 v2.1.0. Always refer to the latest online diagrams and MOF files for the most current version of the model.

## 5.1 Overview

The CDMV2.1 schema can be partitioned into several major conceptual areas:

- Diagnostic services, which include the diagnostic tests and help services
- Capabilities, which identify what the implementation can support
- Settings, which are used to define defaults for the capabilities and specify which capabilities to use on any particular diagnostic test
- Jobs, which are used to monitor and control the execution of diagnostic tests

- Output, which could be either or both diagnostic logs and alert messages
- Concrete Diagnostics Profiles

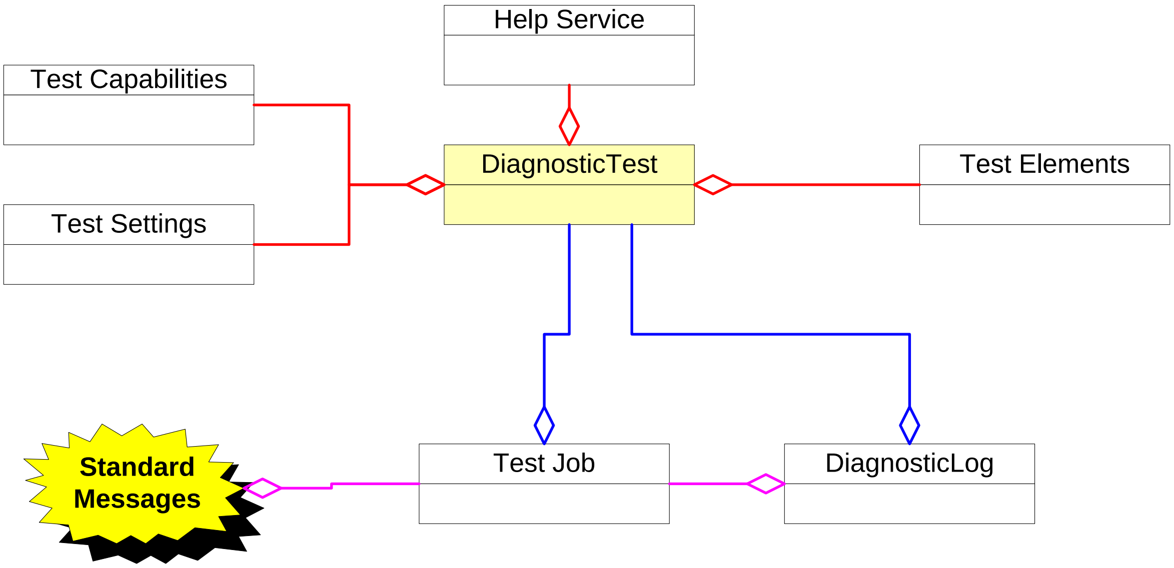



**Figure 1 – Overview of the diagnostics model**

At the center of the model is the diagnostic test service. It provides the operation for invoking tests on the test elements. For example, it might provide a “self-test” on disk drives (the test elements).

The test element would typically be modeled as part of other profiles. For example, the disk drive element might be part of a storage array in a SNIA profile or it might be a disk drive in a SMASH profile.

Associated with the diagnostic test service is a Help Service. This service provides help information for the test operation.

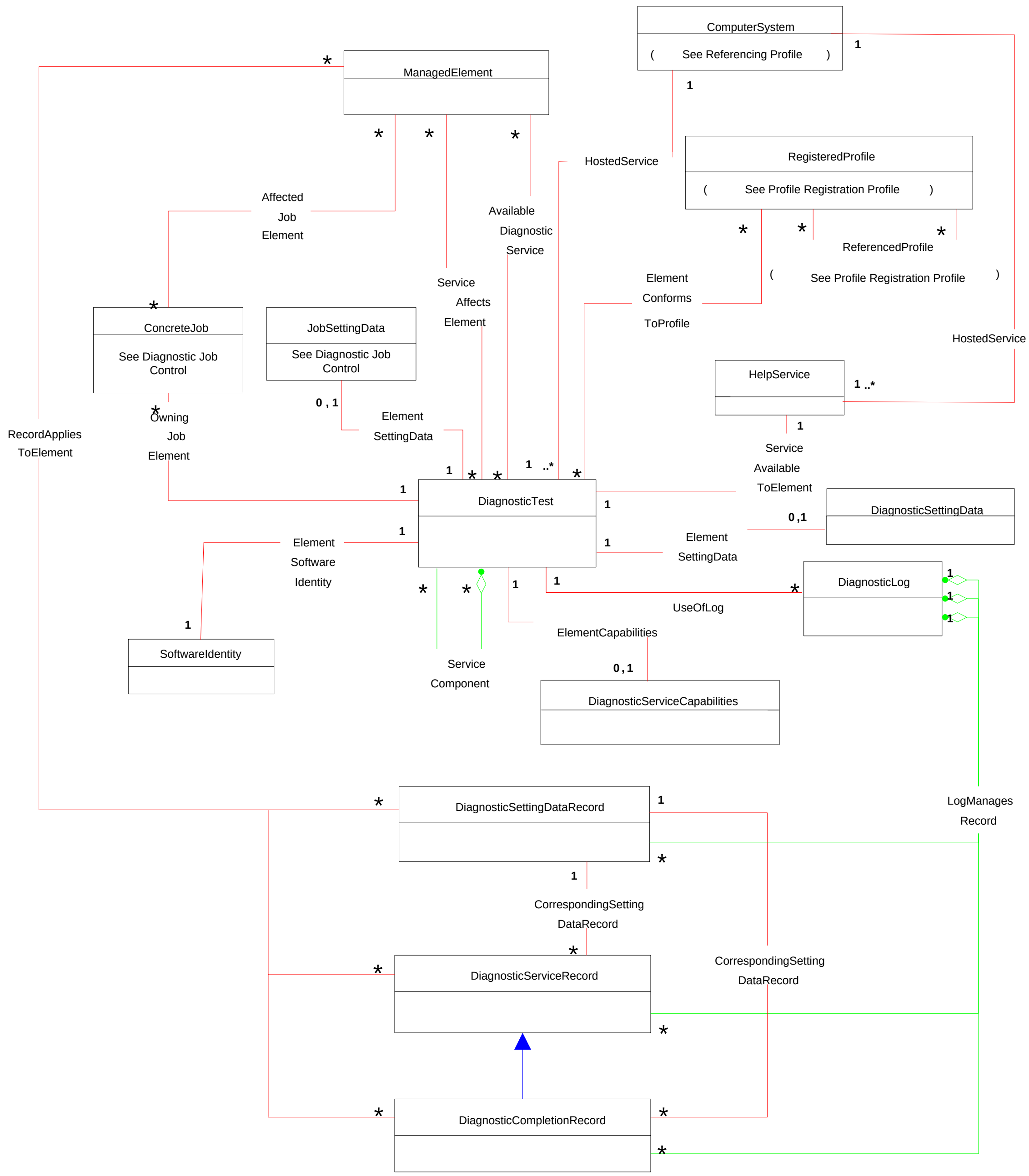
Also associated to the diagnostic test service are test capabilities and default settings for running the test. The capabilities describe the variations that are supported for the test or the job that it creates. For example, there are several service modes that may be supported for a test (HaltOnError, QuickMode, etc.). The default settings identify the defaults that are used by the test if the client application does not specify any settings.

When a test is invoked it will create a job and return control to the client application. This does not mean the test has completed. It merely returns a pointer to the job that is monitoring the progress of the test. The test may generate a log (if requested) for holding the results of the test. The test may also issue standard messages that report on the progress of the test and any errors it may encounter.

When the test (and its job) is completed, the application will be sent a completion standard message, indicating that the job has completed (with or without errors or warnings). It also means that the log has been completely written.

## 5.2 Model components

This clause contains descriptions of the classes in the CIM Schema (CIM 2.34) that support version 2.1 of the diagnostic model.




508

509 **Figure 2 – CDM version 2.1 diagnostics model**

## 510 5.2.1 Services

511 CDM version 2.1 supports two services: the DiagnosticTest service and the HelpService. The
512 DiagnosticTest service supports invocation of a specific diagnostic test. The HelpService supports
513 retrieval of documentation of the test.

#### 5.2.1.1 DiagnosticTestClass

A diagnostic test is modeled with the CIM_DiagnosticTest class. The DiagnosticTest is the only diagnostic service class supported in CDMV2.1.

A diagnostic client uses the properties included in the DiagnosticTest class to determine the general effects associated with running the test. For example, if a test is going to interact with the client, the client needs to be aware of this and inform the user or otherwise be prepared to respond to requests from the test.

A primary function of the diagnostic test (and its associations) is to publish information about the devices that it services and the effects that running the service has on the rest of the system.

The diagnostic service publishes the following information:

- Name and description of the diagnostic test instance
- Characteristics unique to the diagnostic test function
  - For example, “Is Interactive” means that the test interacts with the client application.
- Diagnostic capabilities implemented by the diagnostic test
- Default settings that the diagnostic test applies
- Effects on other managed elements

The diagnostic service (DiagnosticTest) also provides a method for launching the diagnostic processes that implement the test. The RunDiagnosticService( ) method starts a diagnostic test for the specified CIM_ManagedElement (which is defined using the ManagedElement input parameter). How the test should execute (that is, its settings) is defined in a DiagSetting input parameter. The DiagnosticSettings parameter is a string structure that contains elements of the DiagnosticSettingData class. For more information about this class, see clause 5.2.3.1.

The AvailableDiagnosticService: ServiceAvailableToElement class associates the diagnostic service with the managed element that it tests. The managed elements most often targeted by diagnostic services are logical elements such as adapters, storage media, and systems, which are realized by the physical model. The physical model contains asset information about these devices and aggregates them into FRUs.

The ServiceAffectsElement class (not shown in the CDMV2.1 diagram) represents an association between a service and the managed elements that may be affected by its execution. This association indicates that running the service will pose some burden on the managed element that may affect performance, throughput, availability, and so on.

ServiceComponent (not shown in the CDMV2.1 diagram) is an association between two specific services, indicating that the one service (test) may invoke the second service (test) as a component of its test.

DiagnosticServiceCapabilities describes the abilities, limitations, and potential for use of various service parameters and features implemented by the diagnostic service provider. For more information about this class, see clause 5.2.2.1.

Results produced by a test are recorded in an instance of the DiagnosticLog class and linked to the test by an instance of UseOfLog. In addition, the test will produce standard messages in the form of alert indications if clients subscribe to the indications as a means of communicating test results to a client.

#### 5.2.1.2 HelpService class

HelpService was added to fill a need for diagnostic online help. HelpService has properties that describe the nature of the available help documents and a method to request needed documents. Diagnostic services may publish any form of help

CIM_ServiceAvailableToElement should be used to associate the diagnostic service to its help information.

### 5.2.2 Capabilities

Capabilities are “abilities and/or potential for use” and, for the diagnostic model, are defined by the DiagnosticServiceCapabilities and the DiagnosticServiceJobCapabilities classes. Capabilities are the means by which a service publishes its level of support for key components of the diagnostic model. CIM clients use capabilities to filter settings and execution controls that are made available to users. For example, if a service does not publish a capability for the setting “Quick Mode,” the client application might “gray out” this option to the user.

Clients use the ElementCapabilities association from the DiagnosticTest instance to obtain instances of DiagnosticServiceCapabilities and DiagnosticServiceJobCapabilities for the test.

#### 5.2.2.1 DiagnosticServiceCapabilities class

The DiagnosticServiceCapabilities contains properties that identify the capabilities of the DiagnosticTest. These include SupportedServiceModes, SupportedLoopControl, SupportedLogOptions, and SupportedLogStorage. Each DiagnosticTest may advertise its capabilities with an instance of DiagnosticServiceCapabilities to allow clients to determine the options they may specify on the RunDiagnosticService method for invoking the test. The client would specify what they want using the DiagnosticSettings parameter of that method.

##### 5.2.2.1.1 SupportedServiceModes property

This property identifies the service modes supported by the DiagnosticTest. Multiple entries may be provided in the SupportedServiceModes. That is, a test may support none, one, or many of the service modes.

The service modes that may be supported by an implementation include test coverage (PercentOfTestCoverage), accelerated test support (QuickMode), whether you want the test to stop on the first error it encounters (HaltOnError), whether you can set how long results are supposed to be available (ResultPersistence) and whether you want to inhibit destructive testing (NonDestructive).

A client application may choose to use any of the service modes that are advertised by the test implementation in the SupportedServiceModes property. The client application would make its selection using the DiagnosticSettingData class (see 5.2.3.1).

##### 5.2.2.1.2 SupportedLoopControl property

This property identifies the loop controls supported by the DiagnosticTest. Multiple entries may be provided in the SupportedLoopControl. That is, a test may support none, one, or many of the loop controls.

The loop controls that may be supported by an implementation include setting a count of loops (Count), establishing a time limit for the test (Timer) and specifying the test stop after a certain number of errors (ErrorCount).

A client application may choose to use any of the loop controls that are advertised by the test implementation in the SupportedLoopControl property. The client application would make its selection using the DiagnosticSettingData class (see 5.2.3.1).

##### 5.2.2.1.3 SupportedLogOptions property

This property identifies the log options supported by the DiagnosticTest. Multiple entries may be provided in the SupportedLogOptions. That is, a test may support none, one, or many of the log options.

The log options that may be supported by an implementation include log records of several types (e.g., Results, Warnings, Device Errors, etc.). Log records for the log options that are not listed are never logged by the implementation. For a detailed list of log options, see DSP1002 version 2.1.0.

A client application may choose to use any of the log options that are advertised by the test implementation in the SupportedLogOptions property. The client application would make its selection using the DiagnosticSettingData class (see 5.2.3.1).

##### 5.2.2.1.4 SupportedLogStorage property

This property identifies the log storage options supported by the DiagnosticTest. Multiple entries may be provided in the SupportedLogStorage. That is, a test may support none, one, or many of the log storage options. However, in DSP1002 version 2.1.0, only one option is supported. That option is the DiagnosticLog.

An implementation may, however, specify a vendor unique log storage option by including “Other” as a supported log storage option.

#### 5.2.2.2 DiagnosticServiceJobCapabilities class

The DiagnosticServiceJobCapabilities contains properties that identify the job control capabilities of the DiagnosticTest. These include DeleteJobSupported, RequestedStatesSupported, InteractiveTimeoutMax, DefaultValuesSupported, ClientRetriesMax, CleanupInterval, and SilentModeSupported. Each DiagnosticTest may advertise its capabilities with an instance of DiagnosticServiceJobCapabilities to allow clients to determine the options they may specify on the RunDiagnosticService method for invoking the test. The client would specify what they want using the JobSettings parameter of that method.

##### 5.2.2.2.1 DeleteJobSupported

This capability is a Boolean property that indicates whether a client application may issue a DeleteInstance operation on the concrete job that is spawned by the test. If this property is set to FALSE, the DeleteOnCompletion property of the ConcreteJob must always be TRUE. If DeleteJobSupported is TRUE, the DeleteOnCompletion property of the ConcreteJob may be either TRUE or FALSE.

##### 5.2.2.2.2 RequestedStatesSupported

This capability is an array property that identifies the states a client application may request. These should include Terminate and Kill and may include Suspend and Start.

##### 5.2.2.2.3 InteractiveTimeoutMax

This capability identifies the maximum timeout value for interactions with client applications; that is, the maximum time that a test will wait for a client to respond to a request for input or action. This capability only applies to interactive tests.

##### 5.2.2.2.4 DefaultValuesSupported

This capability is a Boolean property that indicates whether an interactive test will accept default values as input on an interactive request to the client application. This capability only applies to interactive tests.

##### 5.2.2.2.5 ClientRetriesMax

This capability identifies the maximum number of retries a test will allow on any one interaction with the client application. An implementation would allow one or more retries to allow a user to correct typographical or other errors on their input.

##### 5.2.2.2.6 CleanupInterval

This capability identifies the time period that the implementation will keep a job defined with DeleteOnCompletion = FALSE. The implementation may delete jobs that have been around longer than the CleanupInterval.

##### 5.2.2.2.7 SilentModeSupported

This capability is a Boolean, that when TRUE, means that the interactive test implementation is capable of running with default values (either the ones defined in the JobSettings parameter or the ones defined in the default JobSettingData). If the value is FALSE, the client application must provide the default inputs as requested by the test.

### 5.2.3 Settings

Settings are classes that are used as input to the RunDiagnosticService method as parameters that control the execution of the test. The RunDiagnosticService includes two parameters to hold this information: DiagnosticSettings and JobSettings. These parameters are string encodings of CIM_DiagnosticSettingData and CIM_JobSettingData classes.

For each of these classes, an implementation may populate instances of the default values. The default CIM_JobSettingData class is required, but a default for the CIM_DiagnosticSettingData is not required. In either case, the range of values that may be specified in the DiagnosticSettings and JobSettings parameters of the RunDiagnosticService method are identified in the CIM_DiagnosticServiceCapabilities and CIM_DiagnosticServiceJobCapabilities.

#### 5.2.3.1 DiagnosticSettingData Class

DiagnosticSettingData is derived from CIM_SettingData and is used to contain the default and run-specific settings for a given test. Diagnostic service providers publish default settings in an instance of this class (associated to the service by a default instance of ElementSettingData), and diagnostic clients create a new instance and populate it with these defaults with, possibly, user modifications. This new setting object is then passed as an input parameter to RunDiagnosticService( ). For all properties except InstanceID and LoopParameter, the values set by a test client in a DiagnosticSettingData object are "qualified" by corresponding properties in DiagnosticServiceCapabilities. If the capabilities do not include support for a setting, the client must maintain the default for that setting. The options that may be selected for the DiagnosticSettings parameter include HaltOnError, QuickMode, PercentOfTestCoverage, LoopControl, LoopControlParameter, ResultPersistence, LogOptions, LogStorage and VerbosityLevel.

##### 5.2.3.1.1 HaltOnError

When this property is TRUE, the test should halt after finding the first error. If the implementation includes a DiagnosticServiceCapabilities instance for the test, HaltOnError should only be set to true when DiagnosticServiceCapabilities.SupportedServiceModes includes “HaltOnError”.

#### 5.2.3.1.2 QuickMode

When this property is TRUE, the test should attempt to run in an accelerated manner by reducing either the coverage or the number of tests performed. If the implementation includes a DiagnosticServiceCapabilities instance for the test, QuickMode should only be set to true when DiagnosticServiceCapabilities.SupportedServiceModes includes “QuickMode”

#### 5.2.3.1.3 PercentOfTestCoverage

This property requests the test to reduce test coverage to the specified percentage. If the implementation includes a DiagnosticServiceCapabilities instance for the test, PercentOfTestCoverage should only be set to true when DiagnosticServiceCapabilities.SupportedServiceModes includes “PercentOfTestCoverage”.

#### 5.2.3.1.4 LoopControl and LoopControlParameter

The LoopControl property is used in combination with the LoopControlParameter to set one or more loop control mechanisms that limit the number of times that a test should be repeated. With these properties, it is possible to loop a test (if supported) under control of a counter, timer, and other loop terminating facilities. If the implementation includes a DiagnosticServiceCapabilities instance for the test, LoopControl should only be set to a value contained in the DiagnosticServiceCapabilities.SupportedLoopControl property.

#### 5.2.3.1.5 ResultPersistence

This property specifies how many seconds the log records should persist after service execution finishes. If the implementation includes a DiagnosticServiceCapabilities instance for the test, ResultPersistence should only be set when DiagnosticServiceCapabilities.SupportedServiceModes includes “ResultPersistence”.

#### 5.2.3.1.6 LogOptions

This property specifies the types of data that should be logged by the diagnostic service.

This capability identifies whether a client may specify the nature of data to be logged by the test. If the implementation includes a DiagnosticServiceCapabilities instance for the test, LogOptions should only be set to values contained in DiagnosticServiceCapabilities.SupportedLogOptions property.

#### 5.2.3.1.7 LogStorage

This property specifies the logging mechanism to store the diagnostic results. If the implementation includes a DiagnosticServiceCapabilities instance for the test, LogStorage should only be set to values contained in DiagnosticServiceCapabilities.SupportedLogStorage property.

#### 5.2.3.1.8 VerbosityLevel

This property specifies the desired volume or detail logged for each log option supported by a diagnostic test. The possible values include Minimum, Standard, and Full. The actual meaning of Minimum, Standard, and Full is vendor specific, but the default is Standard. Full means everything that the implementation supports and Minimum means the minimal amount of information supported by the implementation.

### 5.2.3.2 JobSettingData class

The JobSettingData class is used to specify the default settings for controlling the execution of the test job. The JobSettings parameter of the RunDiagnosticService may contain values that are supported by the DiagnosticServiceJobCapabilities associated with the DiagnosticTest. Clients may encode the values they desire in the JobSettings parameter or let the parameter default to the default instance of the JobSettingData.

The options that may be selected for the JobSettings include DeleteOnCompletion, InteractiveTimeout, TerminateOnTimeout, DefaultInputValues, DefaultInputNames, ClientRetries. and RunInSilentMode.

#### 5.2.3.2.1 DeleteOnCompletion

This property indicates whether the job should be automatically deleted upon completion. If the implementation includes a DiagnosticServiceJobCapabilities instance for the test and CIM_DiagnosticServiceJobCapabilities.DeleteJobSupported is FALSE, the value of CIM_JobSettingData.DeleteOnCompletion must be TRUE. If CIM_DiagnosticServiceJobCapabilities.DeleteJobSupported is TRUE, the CIM_JobSettingData.DeleteOnCompletion may be either TRUE or FALSE.

If DeleteOnCompletion is FALSE, the client is responsible for deleting the job.

#### 5.2.3.2.2 InteractiveTimeout

This interval time property should have a value if the test is interactive (i.e., CIM_DiagnosticTest.Characteristics property contains the value of 3). This value identifies the time the test should wait for a response from a client after asking the client for input.

#### 5.2.3.2.3 TerminateOnTimeout

This property defines the behavior when a client fails to respond within the time interval specified by the InteractiveTimeout on the last request to the client for input.

#### 5.2.3.2.4 DefaultInputValues and DefaultInputNames

The DefaultInputValues (e.g., device identifiers) may be used if the test is interactive and requires inputs from the client (or user). The DefaultInputNames are the names for the values in DefaultInputNames (e.g., the names of the device identifiers). These two properties are arrays and are correlated such that the names match up with the input values. These properties are only relevant when a test is interactive and it will be asking the user for input values.

#### 5.2.3.2.5 ClientRetries

This property indicates the number of times the diagnostic test will prompt the client for the same response after the client fails to invoke the CIM_ConcreteJob.ResumeWithInput( ) or CIM_ConcreteJob.ResumeWithAction( ) method within a specified period of time (InteractiveTimeout) . This property is only relevant when a test is interactive and it will be asking the user for input values or to take actions.

#### 5.2.3.2.6 RunInSilentMode

This property indicates whether the diagnostic test will not prompt the client for responses even though CIM_DiagnosticTest.Characteristics contains the value of 3 (Is Interactive). When the value is TRUE, no prompts are issued. Instead, the diagnostic test will execute using the default values defined in CIM_JobSettingData.

### 5.2.4 Jobs and Job Control

When an invocation of the RunDiagnosticService method is successful (ReturnCode = 0), an instance of CIM_ConcreteJob is created. This class provides a way for the client to monitor the progress of the test.

DSP1002 version 2.1.0 supports job control using the Diagnostic Job Control profile (DSP1119), which is a specialized version of the DMTF Job Control profile (DSP1103). The Diagnostic Job Control is a required component profile of the Diagnostics Profile.

#### 5.2.4.1 Diagnostic jobs

The ConcreteJob that gets created on a successful invocation of the RunDiagnosticService method is associated to the DiagnosticTest that spawned it by the CIM_OwningJobElement association. The ConcreteJob also has a CIM_AffectedJobElement association to the CIM_ManagedElement (e.g., device) on which the test is acting.

The ConcreteJob also has a CIM_HostedDependency association to the system in which the tested device is contained. This allows clients to monitor all the jobs that are active within the system.

The ConcreteJob contains a number of properties of note: DeleteOnCompletion, TimeBeforeRemoval, JobState, and PercentComplete. In addition, there are three methods available to clients for controlling the execution of the job: RequestedStateChange( ), ResumeWithInput( ), and ResumeWithAction( ). The last two methods can be used for interactive tests.

##### 5.2.4.1.1 DeleteOnCompletion

If the DeleteOnCompletion property is TRUE, the job and its related associations will be deleted automatically. The job will be retained until a specified time expires after the completion of the job (see TimeBeforeRemoval in clause 5.2.4.1.2).

If the DeleteOnCompletion property is FALSE, then the client application is responsible for deletion of the job (using the DeleteInstance operation).

The client application that invoked the test can set this property by specifying DeleteOnCompletion in the JobSettings parameter of the RunDiagnosticService method.

##### 5.2.4.1.2 TimeBeforeRemoval

When the DeleteOnCompletion property is TRUE, the TimeBeforeRemoval is the time interval the implementation must wait after the completion of the job before it may delete it.

If the DeleteOnCompletion property is FALSE, this property is ignored.

##### 5.2.4.1.3 JobState

The JobState property identifies the current state of the job. The possible states for a job include the values of 2 (New), 3 (Starting), 4 (Running), 5 (Suspended), 6 (Shutting Down), 7 (Completed), 8 (Terminated), 9 (Killed), 10 (Exception). The job is considered complete if the job states are 7, 8, 9, or 10. The job state of 7 means the job has completed successfully.

##### 5.2.4.1.4 PercentComplete

The PercentComplete property approximates the percentage of the test job that has completed. A percentage of 0 means the test job has not started. A percentage of 100 means the test job has completed. Any percentage in between 0 and 100 means the test job is in progress.

NOTE In some implementations, 50 percent may be the only indication that the job is in progress.

##### 5.2.4.1.5 RequestedStateChange( )

The concrete job can be managed by the client application through the RequestedStateChange operation. This operation may be used to terminate or kill the test job. Terminate means ending the job gracefully. Kill means end the job abruptly, where this may require ending the job without cleaning up. It may also be used to suspend or resume the test job.

#### 5.2.4.1.6 ResumeWithInput( )

The ResumeWithInput operation would be supported for interactive test jobs that require additional input from the client application (that is, the user). The request for input will be made by the test using a standard message. The message will identify the inputs that are required to continue the test. When the user supplies the input to the client application, it would pass those inputs to the test using the ResumeWithInput operation.

#### 5.2.4.1.7 ResumeWithAction( )

The ResumeWithAction operation would be supported for interactive test jobs that require action be taken by the client application (that is, the user). An action might be loading media in a device bay. The request for action will be made by the test using a standard message. The message will identify the actions that are required to continue the test. When the user performs the action and tells the client application, the application would then tell the test using the ResumeWithAction operation.

### 5.2.4.2 Diagnostic Job Control

The Diagnostic Job Control profile is a specialization of the DMTF Job Control profile. It extends the DMTF Job Control profile by adding the DiagnosticServiceJobCapabilities and the JobSettingData classes. It also adds the support for interactive jobs and standard messages as illustrated in Figure 3.

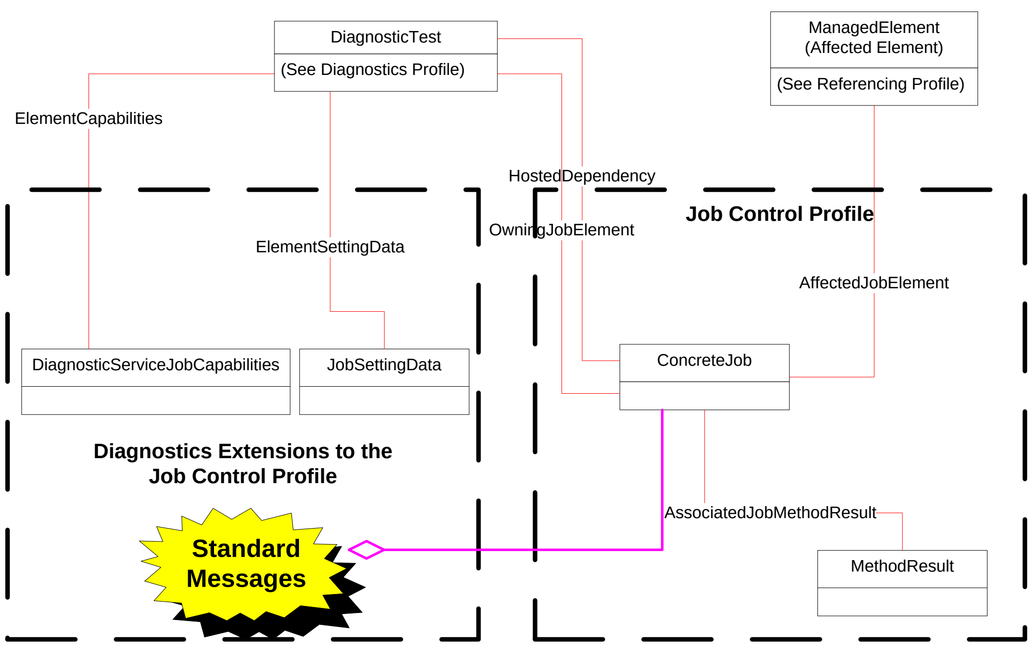



**Figure 3 – Diagnostics Extensions to Job Control**

All other aspects of the DMTF Job Control profile are supported as specified in DSP1103.

### 5.2.5 Output from diagnostics tests

The output of a diagnostic test comes in two forms: the DiagnosticLog and the (Alert Indication) standard messages. The DiagnosticLog output is supported by a test implementation if the LogStorage property of its DiagnosticServiceCapabilities includes the DiagnosticLog option.

Standard messages are alert indications that a test can send during the execution of the test job. In order for a client to receive the alert indications, the client must first subscribe to get the indications. Subscribing to indications is documented in the Indications Profile (DSP1054).

#### 5.2.5.1 Diagnostic logs

If the test implementation supports the diagnostics log and the client has requested a diagnostic log, one instance of DiagnosticLog is created for each invocation of the test. This instance of the DiagnosticLog is associated to the DiagnosticTest instance using the UseOfLog association. Log records are created events that occur during the test and are attached to the DiagnosticLog by using the LogManagesRecord association. This is illustrated in Figure 4.

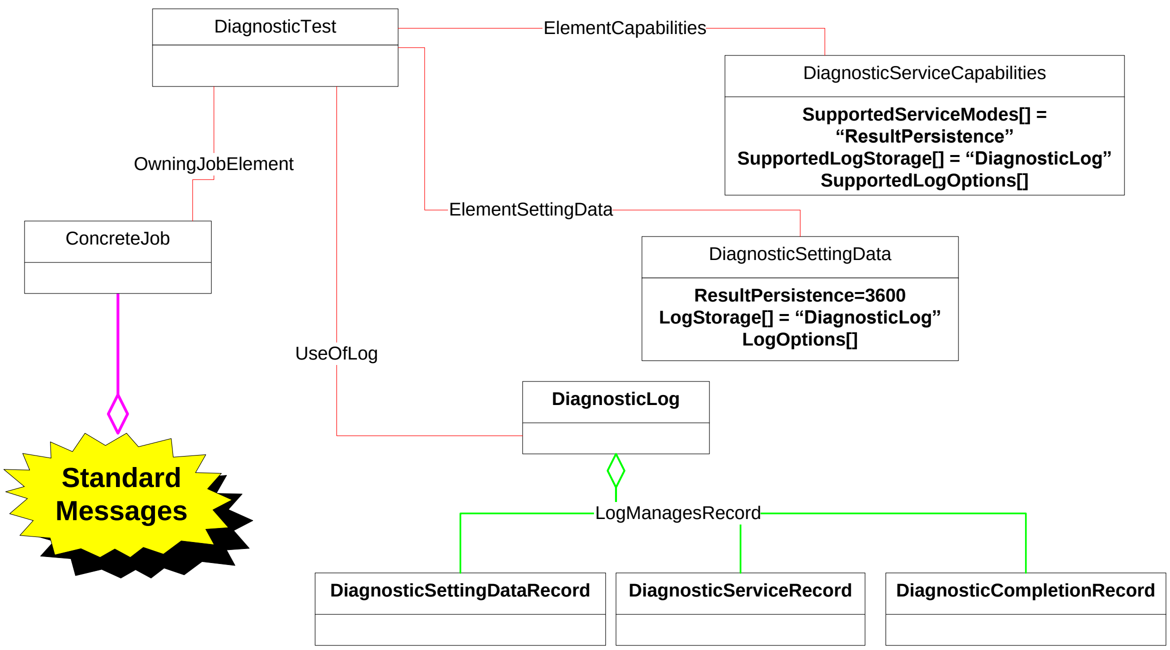



**Figure 4 – Elements for diagnostic logs**

When the diagnostic test is invoked a concrete job is started and a DiagnosticLog is created (assuming DiagnosticSettingData.LogStorage includes “DiagnosticLog”). As the test job executes, standard messages are issued to subscribers and log records are written to the DiagnosticLog. There are three types of records that may be written: a DiagnosticSettingDataRecord, DiagnosticServiceRecords, and a DiagnosticCompletionRecord. The DiagnosticSettingDataRecord identifies the DiagnosticSettingData information that was used, the DiagnosticServiceRecords identify various items that might be logged during the test and the DiagnosticCompletionRecord summarizes the execution status upon completion of the test.

There are three properties in DiagnosticServiceCapabilities and DiagnosticSettingData that pertain to diagnostic logs. If a test is to create a diagnostic log, the

DiagnosticServiceCapabilities.SupportedLogStorage array property should include the enumeration for “DiagnosticLog” and the DiagnosticSettingData.LogStorage must include “DiagnosticLog”.

The DiagnosticServiceCapabilities.SupportedLogOptions array property identifies the types of log records supported by the test and the DiagnosticSettingData.LogOptions array property identifies the record types desired for this execution of the test.

If the DiagnosticServiceCapabilities.SupportedServiceModes array property includes the enumeration for “ResultPersistence”, the DiagnosticSettingData may set the ResultPersistence value. For example, in Figure 4, the ResultPersistence property is set to 3600 seconds (one hour).

#### 5.2.5.2 Diagnostic standard messages

As the test job executes, it may also issue standard messages to the client application by using alert indications reporting the progress, status, errors, and warnings found while running the test. In addition, alert indications are used by the test to communicate directions to client applications for interactive tests. These indications may be subscribed to by the client application so that it can follow what is going on with the test as it executes.

Some test implementations may not have the resources (that is, storage or memory) to keep a diagnostic log. In such cases, the alert indications may be the primary mechanism for the test to report results to the client application. Clients would typically receive the indications and write them to a client log (either in client memory or to a file).

Some alert indications are required to be implemented by the test. For example, completion status messages are required. In addition, if the test is an interactive test, another set of indications are required for handling the interaction with the client application.

An example of exchanges between a client application and the test involving standard messages is illustrated in Figure 5.

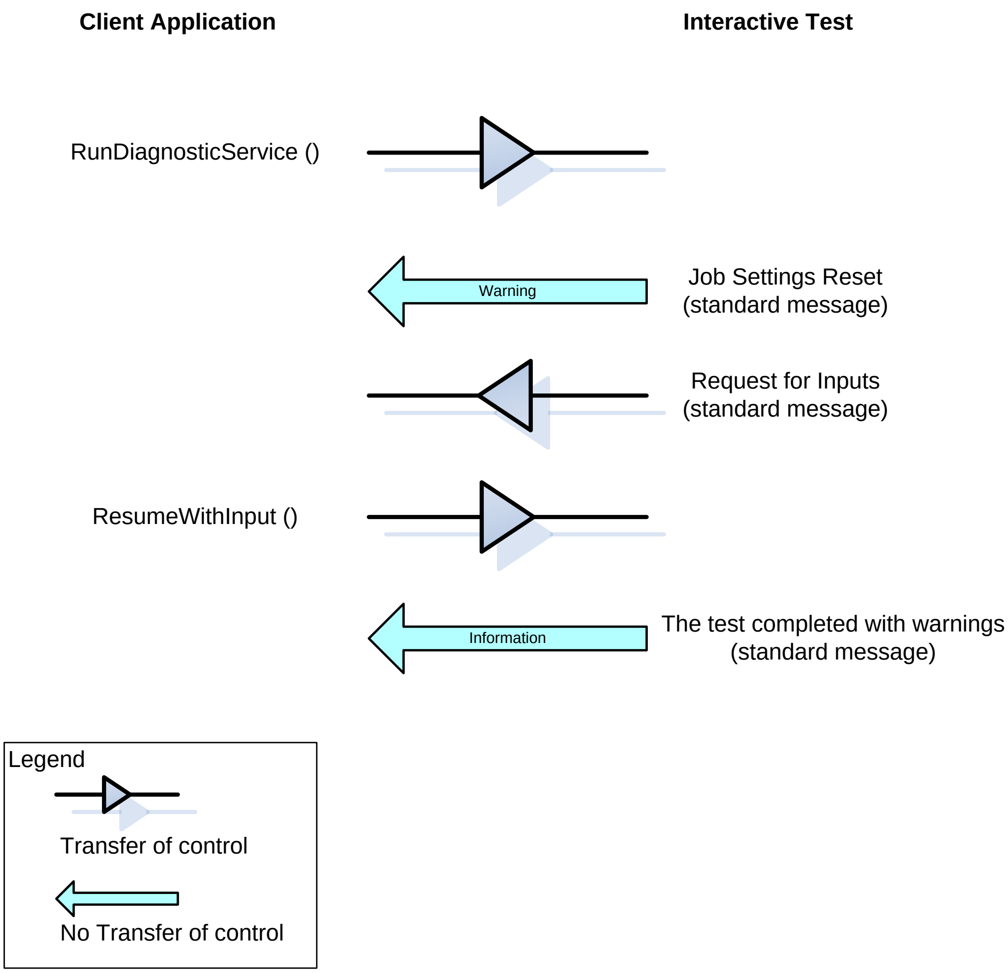



**Figure 5 – Example standard message exchange**

In this example, the client application invokes the test by issuing the RunDiagnosticService ( ) operation. In the process of executing the test, the test discovers that it needs to reset a parameter in the JobSettings passed to it. So the test issues a warning standard message and continues processing the test. When the test needs input from the client application, it issues the “Request for Inputs” standard message. The client application then gets the input from the user of the application and issues the ResumeWithInput ( ) operation. The test then runs to completion and issues a completion standard message indicating that the test was completed with warnings.

### 5.2.6 Concrete diagnostics profiles

The Diagnostics Profile (as defined in DSP1002) is an abstract profile. It is to be used as a pattern for diagnostics implementations and must first be “specialized” to a “concrete” profile. For example, DMTF has defined a number of concrete derivations of DSP1002. These include:

- DSP1104 - Fiber Channel Host Bus Adapter Diagnostics Profile
- DSP1105 - CPU Diagnostics Profile
- DSP1107 - Ethernet NIC Diagnostics Profile

- [DSP1110](#) - Optical Drive Diagnostics Profile
- [DSP1113](#) - Disk Drive Diagnostics Profile
- [DSP1114](#) - RAID Controller Diagnostics Profile

In addition to these concrete profiles, a vendor or organization may define their own concrete profile for diagnostics for a managed element that they manage (see clauses 5.2.6.1 and 5.2.6.2)

Each of these profiles starts from the DSP1002 base (described in this document) and applies the class, functions, and properties to a specific “managed element.” For example, the Disk Drive Diagnostics Profile defines diagnostics support for disk drives. To do this, it extends the definition of DSP1002 by adding DiagnosticServiceCapabilities properties, DiagnosticSettingData properties, and defining specific standard tests that can be run on disk drives. In this case, the “managed element” referenced in this white paper and in DSP1002 is specialized to CIM_DiskDrive.

Figure 6 illustrates how the Disk Drive Diagnostics Profile specializes the abstract Diagnostics Profile.

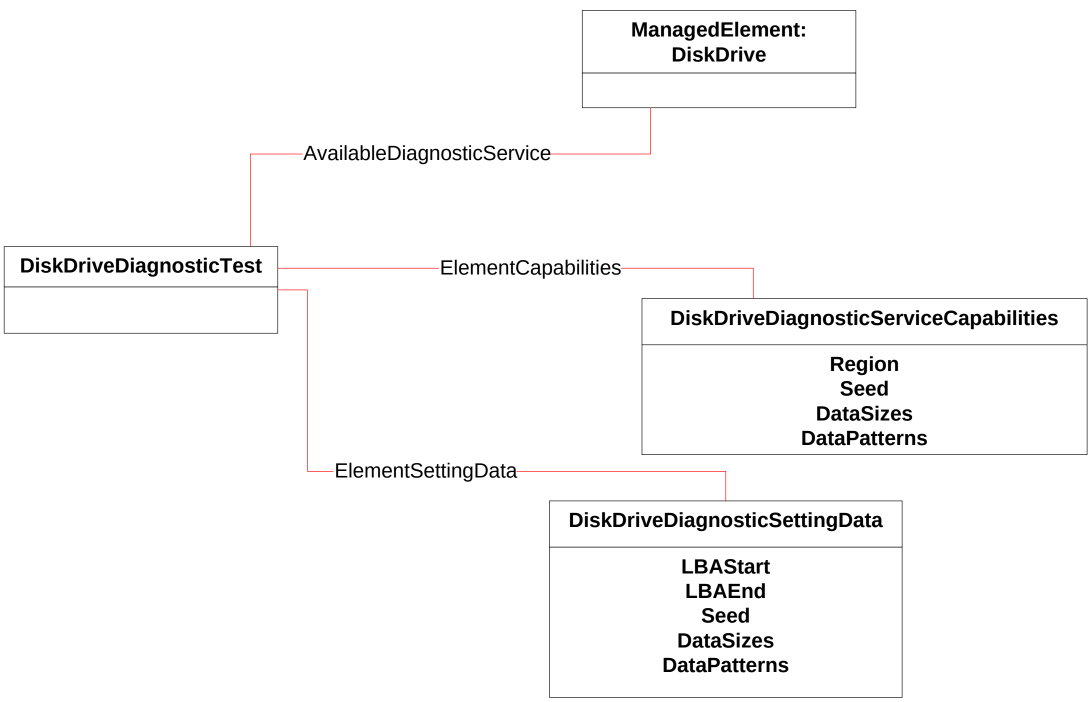



**Figure 6 – Disk Drive specialization of the Diagnostics Profile**

The DiskDriveDiagnosticTest class is a subclass of the DiagnosticTest class. It has all the properties that are in the DiagnosticTest class (such as the Characteristics property). The DiskDriveDiagnosticServiceCapabilities is a subclass of DiagnosticServiceCapabilities. It has all the properties of DiagnosticServiceCapabilities (such as SupportedServiceModes), but it adds four additional properties (shown in Figure 6) that are unique to disk drive testing. The DiskDriveDiagnosticSettingData is a subclass of DiagnosticSettingData. It has all the properties of DiagnosticSettingData (such as HaltOnError), but adds four additional properties (shown in Figure 6) that are unique to disk drive testing. Finally, the managed element that is tested using the DiskDriveDiagnosticTest is, of course, a Disk Drive.

Each different Disk Drive test would have its own instance of DiskDriveDiagnosticTest. The Disk Drive Diagnostics Profile defines 13 tests:

- Short Self-Test
- Extended Self-Test
- Selective Self-Test
- Sequential Read
- Random Read
- Sequential Read-Write-Read Compare
- Random Read-Write-Read Compare
- Sequential Internal Verify
- Status
- Grown Defect
- 4K Alignment
- Power Management
- Performance

Each of these tests would have their own DiskDriveDiagnosticTest instance with their own set of DiskDriveDiagnosticsCapabilities and default DiskDriveDiagnosticsSettingData.

#### 5.2.6.1 Other concrete profiles

DMTF recognizes that the need for diagnostics goes beyond the concrete profiles that are currently defined by DMTF. But the abstract Diagnostics Profile (DSP1002) defines the basic elements that are required for any concrete profile that intends to meet the requirements of CDM. Like the Disk Drive Diagnostics Profile example shown in Figure 6 another organization or a vendor can define their own Diagnostic profile for a new managed element in a similar manner.

#### 5.2.6.2 Extension of concrete profiles

In addition to defining concrete profiles by specializing DSP1002, concrete profiles may also be defined by specializing another concrete profile. For example, if an organization or vendor wants to extend the disk drive diagnostics profile, this can be done by patterning the profile after the DMTF Disk Drive Diagnostics Profile and adding additional properties, and methods, classes or both.

### 5.2.7 Relationship to “Managed Element” profiles

The Diagnostics Profiles have a relationship with the “managed element” profiles of the elements they test. This relationship is primarily with the management of the elements that are tested. To illustrate this point, consider the relationship between the Disk Drive Diagnostics Profile and the SNIA Array Profile as shown in Figure 7.

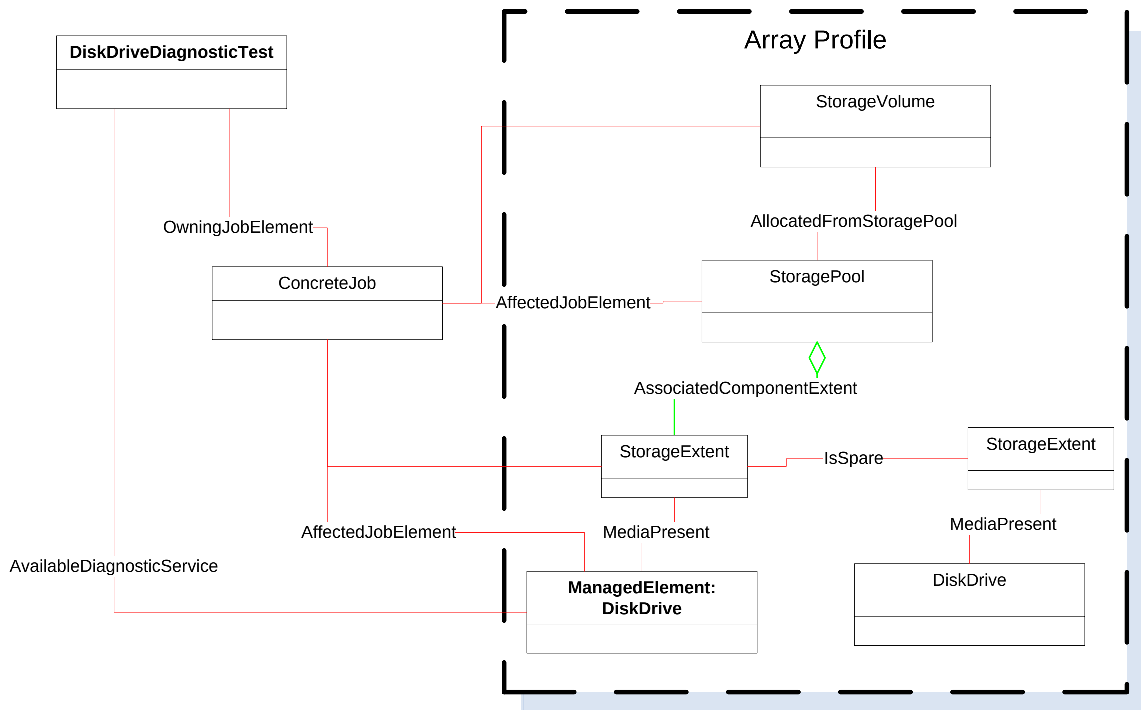



**Figure 7 – Diagnostics and Managed Element Profiles**

In this example, the Disk Drive Diagnostics profile works on the Disk Drive managed element. But the disk drive managed element is just part of an overall Array profile. When the test is invoked on a particular disk drive, a job is created and the job has AffectedJobElement associations to all managed elements that are impacted by the test. This includes a StorageExtent, a StoragePool, and a Volume allocated out of the StoragePool. While the Disk Drive Diagnostics profile will tell you the elements that are affected by the test, it will not tell you how those elements are related. That information is provided by the Array profile (the managed element profile).

Furthermore, if the test results indicate that the disk drive is failing, the Disk Drive Diagnostics profile does not provide the management solution to fix the problem. In the Array example shown Figure 7, the array happens to support a spare drive that can be used to replace the failing drive. Because the sparing function is part of the Array profile, it makes no sense for the Diagnostic profile to duplicate that function. It may indicate that replacing the disk drive is necessary, but it would not provide the function to do the replacement. That would be done by functions in the Array profile.

## 5.3 CDMV2.1 usage

### 5.3.1 Discovery and setup

#### 5.3.1.1 Determining what testing capabilities exist on a system

Client applications can query the CIMOM for the diagnostic services that are associated with the managed elements of interest that are scoped to the hosting system. This system scope could be a computer system, single device, or could represent a network of remotely controlled systems.

To determine the testing capabilities of a system, a client would start from the system (e.g., the ComputerSystem for the system in question) and follow the HostedService association to DiagnosticTest instances.

Each DiagnosticTest will have a name that uniquely identifies the test (e.g., Self-Test). From each DiagnosticTest instance, the client would follow the ElementCapabilities association to obtain the DiagnosticServiceCapabilities and the DiagnosticServiceJobCapabilities instances for the test. These capabilities define what the test is capable of supporting (see 5.2.2).

In addition, by following the AvailableDiagnosticService association from the DiagnosticTest, the client can find the actual managed elements on which the test can work.

NOTE Some tests may invoke other “subtests”. The subtests may or may not be implemented through a diagnostic profile (and may or may not have a DiagnosticTest instance). In any case, the use of these subtests is vendor specific. That is, there is no user control over how the subtests are invoked. For example, a test for a host hardware RAID controller may well invoke individual tests on disk drives in the controller. The test for the controller has settings and capabilities, but not for disk drives. The controller may well execute known tests on the disk drives, but there is no ability for the user of the RAID controller to input settings for the subtests on the disk drives.

#### 5.3.1.2 Configure the service

After the applicable services are enumerated, the client discovers the configuration parameters for each service. (This discovery can occur for all services up front or individually when a service is invoked.)

##### 5.3.1.2.1 Settings

Settings are the runtime parameters that apply to diagnostic services, defined in the DiagnosticSettings parameter (an embedded instance of a DiagnosticSettingData class). Diagnostic services may or may not support all the settings properties, and this support is published using Capabilities (see 5.3.1.2.2).

A diagnostic service should publish its default settings with an instance of DiagnosticSettingData, associated by an instance of ElementSettingData. The client application would traverse the ElementSettingData association (with IsDefault=true) from the DiagnosticTest to the default DiagnosticSettingData. Clients combine these defaults with user modifications (if supported in Capabilities) into an embedded instance of DiagnosticSettingData to be used as the DiagnosticSettings input parameter when invoking the RunDiagnosticService( ) method. Passing a null reference instructs the service to use its default settings.

##### 5.3.1.2.2 Capabilities

Capabilities are “abilities and/or potential for use” and, for the diagnostic model, are defined by the DiagnosticServiceCapabilities class (or one of its subclasses). Capabilities are the means by which a service publishes its level of support for key components of the diagnostic model. CIM clients use capabilities to filter settings and execution controls that are made available to users. For example, if a service does not publish a capability for the setting “Quick Mode,” the client application might “gray out” this option to the user. The user would interpret the “grayed out” option as not available for setting. The client application would not let a user change a grayed out option.

Client applications would use the ElementCapabilities association to traverse from the DiagnosticTest instance to the DiagnosticServiceCapabilities instance for that DiagnosticTest.

##### 5.3.1.2.3 Characteristics

Characteristics[ ] is a property of the DiagnosticTest class that publishes certain information about the inherent nature of the test to the client. It is a statement of the operational modes and potential consequences of running the service. For example, “IsDestructive” indicates that, if this service is started, it will cause some negative system consequences. These consequences can usually be deduced by considering the service, the device upon which the service is acting, and the “affected resources” (see 5.3.1.2.4).

Client applications should examine the Characteristics[ ] array of the DiagnosticTest instance and use this information to determine what the test will or will not do and avoid situations that would be counterproductive to the problem-determination goals. For example, if the Characteristics contains “Is Interactive”, the client application needs to anticipate getting alert requests from the test. Similarly, if the Characteristics contains “Is Destructive”, the client application needs to ensure that data will not be lost by running the test or that no state changes would result from running the test.

##### 5.3.1.2.4 Affected resources

CDM uses the ServiceAffectsElement association to indicate the managed elements affected by the diagnostic service.

Client applications would traverse this association to determine the system consequences of starting the service. The association could be to component elements of the element under test or it could be to elements that are derived from the element under test.

##### 5.3.1.2.5 Dependencies

A service may depend on tests of other components for its successful execution. For example, to test an FC HBA, it may be necessary to run tests on the ports on the HBA. Similarly a test of RAID controller may require tests on the disk drives controlled by the RAID controller. The ServiceComponent association is used to publish these dependencies.

#### 5.3.1.3 Settings protocol

To control the operation of a diagnostic service, a CDM provider must satisfy a number of requirements for supporting the diagnostics schema. For each test, the provider publishes a single instance of DiagnosticServiceCapabilties to indicate what features are selectable in a DiagnosticsSettings parameter. It should provide default settings for the service in an instance of DiagnosticSettingData and link the default settings instance to the diagnostic test instance using the ElementSettingData association.

Any CDM client application can query the CIM server for DiagnosticTest instances. After selecting a test to run, the client should check for its default settings (see clause 5.3.1.2.1) and capabilities (see 5.3.1.2.2) by querying for the ElementSettingData and ElementCapabilities association instances. The client creates an instance of DiagnosticSettings and populates it with the default settings and any modifications made by the user, taking into account the published capabilities for that test.

The RunDiagnosticService( ) method in DiagnosticService can be used to start a diagnostic test. An embedded instance of DiagnosticSettingData is passed as a DiagnosticSettings parameter to the method call. If the DiagnosticSetting parameter is not passed (that is, it is NULL), the CDM provider should use the default setting values.

The diagnostic model uses settings to specify the parameters that are standard to all CIM diagnostic services. The diagnostic settings are never instantiated in the provider. Instead, the client passes test settings to the diagnostic service as a parameter.

When a test's RunDiagnosticService( ) method is called, the test provider may create an instance of DiagnosticLog. The provider then copies each of the properties in the effective DiagnosticSettings parameter into the DiagnosticSettingDataRecord instance associated to the log, thus preserving a record of the settings used for that test execution. An effective DiagnosticSettingDataRecord is what was passed by the client as modified by the provider. When the test has started, a reference to a ConcreteJob instance is returned to the client. The client may then use this reference to monitor the job and the test progress (PercentComplete, JobState).

#### 5.3.1.4 Looping

Properties in the DiagnosticSettingData allow specification of looping parameters to a diagnostic provider. These properties are actually arrays of controls that may be used alone or in combination to achieve the desired iteration effect.

The LoopControlParameter property is an array of strings that provide parameter values to the control mechanisms specified in the LoopControl property. This property has a positional correspondence to the LoopControl array property. Each string value is interpreted based on its corresponding control mechanism. Four types of controls may be specified in the LoopControl array:

- Loop continuously
- Loop for N iterations
- Loop for N seconds
- Loop until greater than N hard errors occur

For example, if a client wants to run a test 10,000 times or for 30 minutes, whichever comes first, it could set both count and timer controls into the LoopControl array to achieve the logical OR of these controls. In another example, if a client wants to run a test 1000 times or until 5 hard errors occur, two elements are set in this array, one of 'Count' and one of 'ErrorCount'. In the LoopControlParameter array, "1000" would be in the first element and "4" in the second element.

If the LoopControl array is empty or null, no looping takes place. Also, if one element is 'Continuous,' the client must determine when to stop the test.

#### 5.3.1.5 Result persistence

Each time a diagnostic test is launched, an instance of DiagnosticLog is created (if SupportedLogStorage indicates some form of log storage). When a log is created, the log is associated to the DiagnosticTest (via the UseOfLog association).

NOTE A job is also created when the test is invoked. The persistence of the log is independent of the persistence of the job. Both the log and job are managed separately.

Some situations (such as abnormal termination) could lead to an accumulation of old, unneeded results. The potential for this type of problem is exacerbated by looping.

In general, diagnostic clients should implement a persistence policy and handle storage of results as needed. Providers should be required to retain results only long enough for clients to secure them. This time can vary, however, depending on the environment in which the testing is being performed and unexpected events that may occur. A setting property allows a diagnostic client to specify how long a provider must retain the DiagnosticLog after the running of a DiagnosticTest. This ResultPersistence property is part of the DiagnosticSettingData class. A provider advertises that it supports the ResultPersistence property in the SupportedServiceModes property of the DiagnosticServiceCapabilities. If it is supported, for each running of a diagnostic test, the client may specify whether and how long a provider must persist the results of running the test, after the test's completion. In typical use, a client makes one of the following choices:

- Do not persist results (ResultPersistence = 0x0): The client is not interested in the results or is able to capture the results prior to completion of the test. The provider has no responsibility to maintain any related diagnostic log after test completion.
- Persist results for some number of seconds (ResultPersistence = <non-zero>): The client needs the results persisted for the specified number of seconds, after which the provider may delete

them. The client may delete the results prior to the timeout value being reached using the DeleteInstance operation on the DiagnosticLog.

- Persist results forever (ResultPersistence = 0xFFFFFFFF): A maximum timeout value prohibits the provider from deleting the referenced diagnostic log. The client is responsible for deleting the log using the DeleteInstance operation on the DiagnosticLog.

NOTE No default timeout value is specified by the profile for this property. However, if the provider publishes a default DiagnosticSettingData, the default value will be in the ResultPersistence property of that instance.

#### 5.3.1.6 LogOptions for typed messages

The DiagnosticSetting.LogOptions property identifies the list of message types that the client could specify. The set of supported message types is extensible; see the DiagnosticSettingData MOF for the most current list. Some examples of types of log options include:

- "Warnings" (value = 5): Log warning messages; for example, 'device will be taken off line', 'test is long-running', or 'available memory is low'.
- "Device Errors" (value = 7): Log errors related to the managed element being serviced.
- "Service Errors" (value = 8): Log errors related to the service itself rather than the element being serviced, such as 'Resource Allocation Failure'.
- "Debug" (value = 14): Log debug messages. These messages are vendor specific.

The CDM provider indicates that it supports various types of messages by setting values in the DiagnosticServiceCapabilities.SupportedLogOptions array. A client then selects what messages it wants captured by listing those types in the LogOptions property of the DiagnosticSettings parameter (an embedded instance of the DiagnosticSettingData class). The log options are independent and may be used in combinations to achieve the desired report. The default behavior is for an option to be off/disabled.

### 5.3.2 Test execution

#### 5.3.2.1 Execute the service

After the client considers all the system ramifications discussed in the preceding clause and chooses a service to run, it starts the service by invoking the RunDiagnosticService( ) method of the DiagnosticTest class. The diagnostic service provider receives settings and a reference to the managed element object to be used in running the service. If successful, the provider creates an instance of ConcreteJob, and returns a reference to it.

##### 5.3.2.1.1 Starting a test

A diagnostic test job is launched in the following manner:

1. When its RunDiagnosticService( ) method is called and it passes basic parameter checks, the diagnostic service provider creates an instance of ConcreteJob, creates a globally unique InstanceID key (see clause 5.3.2.1.1), and returns a reference to the job object as an output parameter.
    - The test is controlled by the DiagnosticSettings parameter (an embedded instance of a CIM_DiagnosticSettingData class).
    - The job is controlled by the JobSettings parameter (an embedded instance of a CIM_JobSettingData class).

2. The diagnostic service provider creates the associations OwningJobElement and AffectedJobElement so that the client can identify which diagnostic service owns the job and what effects the job will have on various managed elements.
3. When the job is completed, the client will either have or can retrieve the results of the test. See 5.3.2.6 for how to test for job completion and 5.3.3 for determining the results of the test.

#### 5.3.2.2 Monitor and control the test

The client can use the job object to monitor and control the running of the test with the following properties and methods:

- ConcreteJob.JobState—Property that communicates the current state of the job. Values are "New", "Starting", "Running", "Suspended", Shutting Down", "Completed", "Terminated", "Killed", "Exception", and "QueryPending".
- ConcreteJob.DeleteOnCompletion – Property that identifies whether the job will be deleted upon completion of the test (plus the TimeBeforeRemoval interval).
- ConcreteJob.TimeBeforeRemoval – The time interval between job completion and deletion of the job when DeleteOnCompletion is in effect.
- Job.PercentComplete—Property that communicates the progress of the job.
- Job.ElapsedTime—The time interval that the job has been executing or the total execution time if the job is complete.
- ConcreteJob.RequestStateChange( ) –Method used to change the JobState. Options are "Start", "Suspend", "Terminate", and "Kill".
- ResumeWithInput( ) – Method used to communicate that the user has taken an action requested by an interactive test.
- ResumeWithAction( ) – Method used to communicate that the user has taken an action requested by an interactive test.

#### 5.3.2.3 Standard messages

If a client application has subscribed to the alert indications for a test, it will get alert indications (standard messages) as the test executes. These messages report events that occur during the test. Ultimately, there will be an alert indication that indicates that the test was completed successfully, with warnings, or with errors.

For tests that do not support extensive logging, the client should subscribe to the indications to collect information about the test.

#### 5.3.2.4 Interactive tests

Some tests will be interactive. That is, the test will request additional input from the user (client application) to continue with the test. This might be connecting a device or inserting or removing media into (or from) a device bay.

A user can determine if a test is interactive by inspecting the Characteristics property of the DiagnosticTest instance for the test. This is a string array property. If the string array includes the value “3” (“Is Interactive”), the application should be prepared to receive the alert indications that request actions or inputs. If the value “3” (Is Interactive) is not present in the Characteristics array, the test will never make interactive requests (that is, the test is not interactive).

To receive these indications, the client must be subscribed to the DIAG34 and DIAG35 alert indications. These are the standard messages requesting inputs or actions. In addition, the client application should also subscribe to the DIAG9 (Test continued after last interactive timeout using default values), DIAG48 (Test continued after an interim interactive timeout) and DIAG49 (Test terminated after an interactive timeout) standard messages. These report events related to interactive testing.

For a complete list of standard messages for diagnostics, see DSP8055 (the DMTF Diagnostics Message Registry).

#### 5.3.2.5 Complete the test

A client can use the preceding controls to terminate a test job or the test job may be completed normally when its work is done. The client monitors the controls to determine when the test job is completed.

The outcome of running a test is generally presented as a series of messages and data blocks that the client can use in the problem-determination process. In CDM, the DiagnosticLog class is used for data kept by the provider. Test providers instantiate subclasses of DiagnosticRecord for logging data that the test job returns. These are aggregated to a log with the LogManagesRecord association. A client may attempt to read these records by traversing the UseOfLog and LogManagesRecord associations.

Messages are sent to the client as AlertIndications as they happen during the test. The client should subscribe to the alert indications to receive them. After the client receives an alert indication, it may record the information provided in a client record store because the provider- maintained log has limited capacity and lifespan.

#### 5.3.2.6 Checking for test completion

Client applications should be checking for the completion of the test job. All diagnostic tests are run as jobs and are under job control after the client gets a zero return code from the RunDiagnosticService method invocation.

The ConcreteJob instance for the test job has two properties that can be checked. When a job has completed, the JobState property will be 7 (Complete). The OperationalStatus will contain 2 (OK) and 17 (Complete) if the job completed successfully. The OperationalStatus will contain 6 (Error) and 17 (Complete) if the job encountered an error.

An OK completion or completion with an error does not necessarily tell the client what the test results are. For this, the client can either check the logs for the job or subscribe to the appropriate alert indications. Logging may or may not be supported, but alert indications will always be supported. The alert indications that tell the client that the test has completed are:

- DIAG0 - The test passed
- DIAG3 - The device test failed
- DIAG4 - The test completed with warnings
- DIAG44 - The test did not start
- DIAG45 - The test aborted

NOTE The DIAG45 message would be sent if the test was terminated or killed. Other DIAG alert messages will identify whether the job was killed or terminated and whether the action was taken by the client or the server. The JobState will also identify whether the job was terminated or killed.

Other alert indications would provide details about the conditions encountered during the test.

NOTE To receive the alert messages, the client must be subscribed to the alert indications. Minimally, the client should subscribe to the completion status messages shown above.

### 5.3.3 Determining the results of a test

When the RunDiagnosticService is invoked a zero return code indicates that the test job has been created and is executing the test. The results of the test are communicated in two ways:

1) Alert indications
2) Diagnostic log

Support for the log is optional. Some profile implementations run in limited storage environments and cannot support maintaining a log. As a result, alert indications should always be supported by profile implementations. A client can determine whether a log is supported via the SupportedLogStorage property of the CIM_DiagnosticServiceCapabilities instance associated to the DiagnosticTest.

#### 5.3.3.1 Alert indications

A client can follow the execution of a test by subscribing to alert indications generated by the test. As the test runs, it will generate the alert indications to any listener that is subscribed to the alerts.

With alert indications, a client can react to events as they occur. This may be as simple as writing its own log of events generated by the test or it could be responding to a request for input or action made by the test (for interactive tests).

Alert indications may be standard alert indications (documented in the profile) and they may include vendor unique indications. The standard alert indications provide a standard way of reporting events generated by the test.

#### 5.3.3.2 Diagnostic log

If the test supports logging of information associated with the test, a log will be created for the test run. This log will be associated to the DiagnosticTest instance from which the test was invoked. It is important to note that one log is created for each invocation of the test.

The InstanceID of the DiagnosticLog does not identify which invocation of the test that the log records. However the individual log records contain InstanceIDs that include the InstanceID of the ConcreteJob representing the particular invocation of the test. Specifically, the InstanceID of a log record is the InstanceID of the ConcreteJob with a suffix of the “sequence number” of the record.

While there are certain properties in the log record that are standard, most of the information about the test event is vendor specific. The client should refer to vendor documentation on the contents on log records.

There are two special log records that should be included in any given log. The first is a DiagnosticSettingDataRecord, which reports the DiagnosticSettings values that were used with the test. The second is the last log record, which is the DiagnosticCompletionRecord that reports the results of the test.

### 5.3.4 General usage considerations

#### 5.3.4.1 Flushing out errors early

CDM supports testing at any stage of the life cycle of components. It is important to flush out errors early in the life cycle of a component. The earlier errors are discovered, the less it costs to replace or repair the component.

Tests for system development should be designed to exercise the functions of the component to ensure the expected results are produced. Any errors detected in this phase of the life cycle will reduce or eliminate redesign and rework during manufacturing.

Tests for manufacturing should be designed to validate that all functions of the component are operating properly. These tests are particularly useful for components that are OEMed to system integrators for verifying that the components being shipped are working properly. This verification reduces the number of returned components and enhances customer satisfaction.

Tests designed to work at the OEM integrators shop should be designed to verify that the component was not damaged in transit. These tests would be a variation of the self-test to ensure that everything is in working order. For example, for disk drives, this test is called a “conveyance test.”

Tests defined for operation in the customer’s system environment should be designed to report on the health of the component, whether the component is about to fail and whether or not the component should be replaced or repaired. Specifically, the test should help customers isolate failing components. Additional tests may also be made available to service personnel to help in this area.

#### 5.3.4.2 Independent testing of components

Some components are designed to be tested “outside” of a production or system environment. This is to accommodate testing in the manufacturing environment or in acceptance testing by a system integrator. Using CIM and CDM, manufacturing and acceptance testing can be achieved in one of two ways:

1) Providing TCP/IP access to the component that has a CIM Server
3) Providing another access protocol (via interfaces provided, such as SCSI or Wi-Fi)

Either one of these techniques may be used to invoke the test from a client that resides outside the device.

#### 5.3.4.3 Interaction of tests with their environment

Test results can be affected by the environment in which they are running. In many cases, tests will run when other concurrent activity is present. To prevent concurrent activity, the user should quiesce the system before running the test to avoid “outside influences” on the test.

In some cases, the test may actually tell the user that it cannot run due to current conditions. In these cases, the test job will generate an alert message (DIAG12), which indicates that the job was not started. That alert message will also provide a reason for why the job was not started. Some of the reasons might include:

- Element already under test
- Too many jobs running
- Test disabled
- Element disabled
- Element in recovery
- Resources are inadequate to run job

The alert message provides the user with information necessary to change the conditions to allow the test to run. When the user gets a DIAG12 alert message, no job will be created and the user must clear the condition and re-run the test.

NOTE To receive the alert message, the client must be subscribed to the DIAG12 alert indication.

#### 5.3.4.4 Testing degraded elements

In CIM models for management, many of the key elements in the management domain will report status. In SMI-S, for example, the OperationalStatus property is used extensively to report the status of managed elements. Users can determine testing required based on the status information.

If an element is reporting an OperationalStatus of “Stressed” or “Degraded”, various tests might be run to determine the reason for the status. A self-test might be run to determine the overall health of the element. Performance tests might be run to determine performance problems.

If an element is reporting an OperationalStatus of “Error”, the client should run tests to determine why the element is reporting an error. This investigation might start with a self-test, but may involve more pointed tests after reviewing the results of the self-test.

If an element is reporting an OperationalStatus of OK and nothing else, testing on that element would only be done to verify the element is operating properly. Typically, this might be a self-test.

### 5.3.5 Development usage considerations

#### 5.3.5.1 Provider development with common infrastructures

The infrastructure for developing WBEM based agents for the management of systems and devices can be obtained from several sources. Most of these come with SDKs (system development kits). Some of the more common sources of WBEM software include:

- WBEM Solutions J WBEM Server (See WBEM Solutions.)
- OpenPegasus (See OpenPegasus.)
- Windows Management Instrumentation (See WMI.)

#### 5.3.5.2 Client development with common infrastructures

In addition to infrastructures for developing management agents, most of the sources also provide client libraries for accessing WBEM management agents. Such libraries take WBEM requests and build the actual xml messages that are sent to the management agents. The infrastructures identified in the previous clause also provide client libraries for accessing WBEM servers.

In addition, another source for a client library is SBLIM (see SBLIM).

### 5.3.6 Correlation of logs and jobs

Figure 8 illustrates an example of a test that is run multiple times and the resulting logs and jobs.

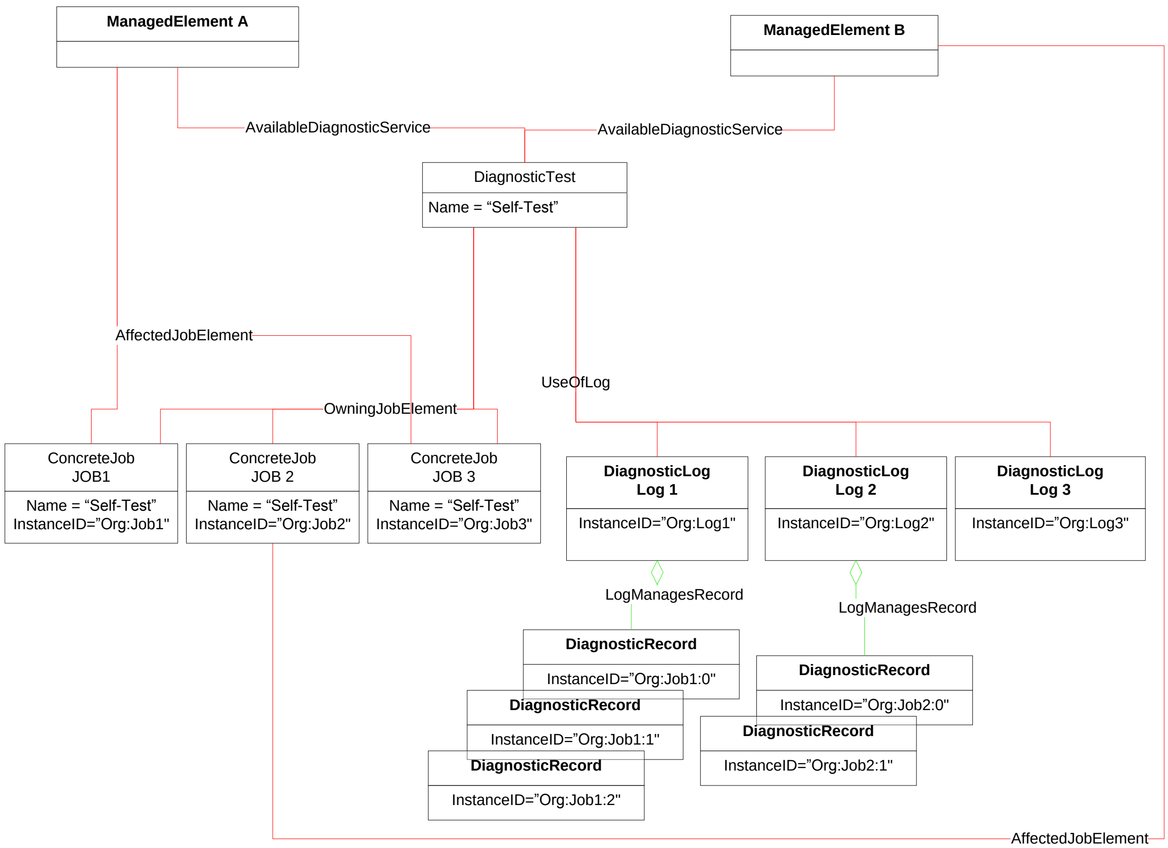



**Figure 8 – Jobs and logs**

The process flows as follows:

1) The client queries for available services and decides to run three instances of a service on two managed elements.

2) The client invokes RunDiagnosticService( ) on ManagedElement A with the appropriate settings and receives a reference to Job1 (InstanceID = ”Org:Job1).

3) The service is started, and Job1 is used for client/service communication and a new log (Org:Log1) is created.

4) Similar actions take place for the second ManagedElement instance (ManagedElement B) and Job2 (InstanceID = Org:Job2) and a second log (Org:Log2) is created.

   Note that it is an implementation detail whether there are two instances of the service provider running or the provider is able to handle multiple requests of this kind.

5) Two keyed jobs are running (Org:Job1 and Org:Job2), generating keyed log records. The next clause addresses these keys and how they should be constructed.

6) After a service job is complete, the job associated with it may be deleted (if DeleteOnCompletion is TRUE). The results of the tests are obtained from the log and its aggregated DiagnosticServiceRecords.

7) The client invokes RunDiagnosticService( ) on the first managed element (ManagedElement A) and a third job (Org:Job3) and a third log (Org:Log3) is created.

#### 5.3.6.1 CDM key structure

Keeping object references distinct is critical in this environment. Object references include key values for uniqueness, and a convention for key construction is often required to guarantee this uniqueness.

##### 5.3.6.1.1 ConcreteJob key

The ConcreteJob class contains a single opaque key, InstanceID. The MOF description provides the following guidance for its construction:

*“The InstanceID must be unique within a namespace. In order to ensure uniqueness, the value of InstanceID SHOULD be constructed in the following manner: <Vendor ID><ID>. <Vendor ID> MUST include a copyrighted, trademarked or otherwise unique name that is owned by the business entity or a registered ID that is assigned to the business entity that is defining the InstanceID. (This is similar to the <Schema Name>_<Class Name> structure of Schema class names.) The purpose of <Vendor ID> is to ensure that <ID> is truly unique across multiple vendor implementations. If such a name is not used, the defining entity MUST assure that the <ID> portion of the Instance ID is unique when compared with other instance providers.”*

##### 5.3.6.1.2 DiagnosticRecord key

The DiagnosticRecord class has a single key, InstanceID. It is constructed to include the ConcreteJob InstanceID key. In addition, the DiagnosticRecord InstanceID includes a sequence number as a suffix.

It is further specified in the Diagnostics Profile Specification that:

*To simplify the retrieval of test data for a specific test execution, the value of InstanceID for CIM_ConcreteJob is closely related to the InstanceID for the subclasses of CIM_DiagnosticRecord.*

*CIM_DiagnosticRecord.InstanceID should be constructed by using the following preferred algorithm:*

*<ConcreteJob.InstanceID>:<n>*

*<ConcreteJob.InstanceID> is <OrgID>:<LocalID> as described in CIM_ConcreteJob, and <n> is an increment value that provides uniqueness. <n> should be set to 0 for the first record created by the test during this job, and incremented for each subsequent record created by the test during this job. Each new test execution will reset the <n> to 0.*

##### 5.3.6.1.3 Correlation of jobs and logs

The client application can determine which log belongs to which job by inspecting the diagnostic records in the log. The first portion of the InstanceID for the record is the InstanceID of the job. The second portion of the InstanceID is the sequence number of the diagnostic record. This can be seen in the example in Figure 8.

# 6 Future development

At the time of this writing, CDM has defined and published in two versions: CDMV1 and CDMV2. CDMV2 continues to extend and enhance the functions introduced CDMV1. The futures described in this clause may be defined in later releases of CDMV2 if they are backward compatible with the rest of CDMV2. Other enhancements will be introduced into CDMV3.

## 6.1 Functions for reporting on affected elements

The CDM tests identify the affected job elements. This includes identifying the affect that the test has on the affected element. However, it does not identify the affect a failure on the component under test has on the affected elements. There are two approaches to address this need: Tests on higher level logical elements and diagnostic functions on the failed element.

### 6.1.1 Tests on higher level logical elements

In many ways, this approach is preferred. By exercising a test (e.g., a self-test) on the affected logical elements, the user can determine the affect the failing component has on the logical element. Often this can be a more precise assessment of the situation presented to the affected element.

### 6.1.2 Diagnostic functions on the failed element

An alternative approach is to offer diagnostic (reporting) functions on the failed component. In this approach, a diagnostic report function is invoked with the failing component and the error it is producing as its inputs. The function then would assess the logical elements that would be impacted and the nature of the impact.

This can be useful in determining the scope of the problem presented by the failing component. However, it may still be necessary to run a self-test on each of the affected elements to determine the actual impact.

## 6.2 Reporting of available corrective actions

Some components may have self-correcting functions when errors are detected. However, sometimes corrective action requires user (or service) participation. On the whole, such repair actions fit within the context of the failing element. Corrective actions that involve actions on affected elements would be outside the scope of the repair functions on the failing element.

As a simple example, say a disk drive exhausts it “spare sectors” and can no longer support its stated capacity. If the repair action is to reduce the capacity of the drive, this would be a repair function on the disk drive. If the repair action is to replace the failing drive with a spare and reconstruct the data for the drive on the spare, this is a repair action on an affected element (e.g., the RAID group). The former action is a repair action on the failing component (the disk drive). The latter is a repair action on a higher level affected element (e.g., the RAID Group).

The proposed enhancement would be for a repair function that could be executed on the appropriate element (in the example, either the disk drive or the RAID group). The repair function would identify the desired repair action, the element to be repaired, and any inputs needed to affect the repair. An example of “any inputs” would be the identification of the spare drive to use to fix a RAID group.

## 6.3 Continued integration with initiatives

The diagnostic work in the DMTF has been focused on defining diagnostics for two initiatives: SMASH and SNIA. The work with both SMASH and SNIA elements will continue.

In the case of work with SMASH, the focus will be on completing diagnostic profiles for components of a system. This is expected to include diagnostics for Fans, System Memory, Sensors, and Power Supplies. In addition, as new functions are introduced to the overall architecture, existing diagnostics for components like CPUs, FC HBAs, and disk drives will be updated to incorporate the new functions.

In the case of work with the SNIA (and SMI-S), the focus will be on adding diagnostics for more components, like ports, and higher level logical elements, like storage pools and storage volumes. Like the SMASH work, as new functions are added to the CDM architecture, components in SMI-S will be updated to incorporate those functions.

Note that some diagnostic profiles will be supported by both SMASH and SMI-S (such as fans, sensors, and power supplies).

## 6.4 Integration of the RecordLog profile

The CDM architecture defines a DiagnosticLog and a set of classes and associations that support logging of test results. This is independent of the DMTF Record Log profile. To facilitate standardization of logging functions, future versions or releases of CDM (specifically DSP1002) will incorporate the DMTF Record Log profile.

## 6.5 Improvements to test reporting

As users and clients gain more experience with diagnostic tests, it is anticipated that improvements will be required to satisfy some of their needs in the area of reporting. This could be additional log records, additional alert indications, and possibly additional classes.

An example of additional log records might be log records that record the information conveyed in the alert indications (a new LogOptions enumeration).

An example of adding additional classes is persistent summary results of a test associated to the tested element. That is, a log is transient and will disappear after the client has had a chance to retrieve its information. The persistent record would be a summary of test record that would be retained until deleted by the client. The record would be associated to the tested element.

## 6.6 Improved reporting of testing capabilities

The ability to determine test capabilities, as documented in 5.2.2and 5.3.1.1, covers the basic needs for reporting tests and the capabilities of the tests. However, there are areas where this could be improved upon.

One area is the identification of “subtests” supported by a test. For example, a self-test will typically run a number of “subtests” to confirm proper functioning of an element. But the subtests are not identified. This will become more important as we expand CDM to cover diagnostics for logical elements.

Another area is simplified reporting of tests and elements that support tests for a system. While this can be discovered (see 5.3.1.1), it is a multiple operation process to discover everything in a system that is covered. A future release of CDM might offer a method for retrieving the information via a single method call.

## 6.7 Testing for logical elements

The current CDM functions are oriented toward physical elements (such as field replaceable units). However, to be useful in a more general health and fault management environment, the diagnostic functions need to encompass logical elements that are affected by the physical elements. Any element that reports some property for of health status (such as OperationalStatus) would be a candidate for applying diagnostic testing.

For example, a storage volume on an array subsystem reports OperationalStatus. This status might typically be affected by the status of the disk drives on which it stores its data or the ports used for accessing the disk drives.

CDM support for logical elements is envisioned to include:

- Improved reporting of “subtests” on the elements on which the logical element is based
- Improved logging to distinguish entries that are attributed to subtests
  - This could be separate logs for subtests or log record information that identifies the subtest.
- Improved alert indications to distinguish alerts associated with a subtest
  - This might be identification of the “super test” for a subtest alert indication.

## 6.8 Enhanced reporting of affected job elements

The AffectedJobElement association identifies the effects a test has on elements related to the element under test. However, AffectedJobElement is transient and only reports on the effect of running the test.

Another interesting question is what effect the status of an element has on other elements. In particular, if a test discovers that an element (such as a disk drive) is in an error state, what storage volumes are impacted by the drive in error? Storage volume based on the disk drive would be affected job elements, but the ElementEffect might be “Performance Impact” and the AffectedJobElement goes away when the job goes away.

What remains after the job goes away is the error state in the disk drive and some sort of degraded or error state in the storage volumes. But the linkage between the failing disk drive and the affected storage volumes is gone.

One answer could be the RelatedElementCausingError association. This is an association called for by the SMI-S Health and Fault Management design. This could be populated to identify elements (such as storage pools) that are degraded due to failures in other elements (such as disk drives). But there are limitations to what can be reported using the RelatedElementCausingError association.

Another approach would be a method that reports on the nature of the relationship (such as “Package Redundancy degraded”) and identifies possible corrective actions (such as “apply spare disk” or “replace disk”).

## 6.9 Applying security to CDM functions

DMTF has defined a set of security profiles for defining who is authorized to certain functions defined in CIM models. CDM will look into defining how the security profiles should be applied to CDM functions. This would require adding security profiles (e.g., Identity Management and Role Based Authorization) to the related profile list in DSP1002.

End of document

# ANNEX A
(informative)

## Change log

| Version | Date | Description |
|---|---|---|
| 1.0.0 | 2004-12-14 | The first version of the Diagnostic Model Whitepaper (12/14/2004), based on CIM 2.9. |
| 2.0.0 | 2015-04-14 | The whitepaper updated for CDM Version 2.1 and CIM Schema 2.34 (3/2/2015). |

# Bibliography


DMTF DSP1002, *Abstract Diagnostics Profile 2.0,* http://dmtf.org/sites/default/files/standards/documents/DSP1002_2.0.0.pdf

DMTF DSP1002, *Abstract Diagnostics Profile 2.1*, http://dmtf.org/sites/default/files/standards/documents/DSP1002_2.1.0a.pdf

DMTF DSP1054, *Indications Profile 1.2.1*, http://dmtf.org/sites/default/files/standards/documents/DSP1054_1.2.1.pdf

DMTF DSP1103, *Job Control Profile 1.0,* http://dmtf.org/sites/default/files/standards/documents/DSP1103_1.0.0_0.pdf

DMTF DSP1104, *FC HBA Diagnostics Profile 1.0,* http://dmtf.org/sites/default/files/standards/documents/DSP1104_1.0.0.pdf

DMTF DSP1105, *CPU Diagnostics Profile 1.0.1,* http://dmtf.org/sites/default/files/standards/documents/DSP1105_1.0.1.pdf

DMTF DSP1107, *Ethernet NIC Diagnostics Profile 1.0,* http://dmtf.org/sites/default/files/standards/documents/DSP1107_1.0.0.pdf

DMTF DSP1110, *Optical Drive Diagnostics Profile 1.0,* http://dmtf.org/sites/default/files/standards/documents/DSP1110_1.0.0.pdf

DMTF DSP1113, *Disk Drive Diagnostics Profile 1.0*, http://dmtf.org/sites/default/files/standards/documents/DSP1113_1.0.0.pdf

DMTF DSP1114, *RAID Controller Diagnostics Profile 1.0*, http://dmtf.org/sites/default/files/standards/documents/DSP1114_1.0.0.pdf

DMTF DSP1119, *Diagnostics Job Control Profile 1.0*, http://dmtf.org/sites/default/files/standards/documents/DSP1119_1.0.0b.pdf

DMTF DSP8055, *Diagnostic Message Registry 1.0,* http://schemas.dmtf.org/wbem/messageregistry/1/dsp8055_1.0.0c.xml

*CIM Schema* at http://www.dmtf.org/standards/index.php

CMPI, *Common Management Programming Interface Issue 2.0* https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=12078

JSR48, *Java Specification Request 48* https://jcp.org/en/jsr/detail?id=48

*OpenPegasus* from The Open Group https://collaboration.opengroup.org/pegasus/index.php

SBLIM, *Standards Based Linux Instrumentation for Manageability* http://sourceforge.net/projects/sblim/

SMI-S, *Storage Management Initiative Specification* http://www.snia.org/tech_activities/standards/curr_standards/smi

SMASH, *Systems Management Architecture for Server Hardware* http://dmtf.org/sites/default/files/SMASH%20Overview%20Document_2010.pdf

WBEM Solutions website
http://www.wbemsolutions.com/index.php?p-id=h

WMI, *Windows Management Instrumentation from MicroSoft*
http://msdn.microsoft.com/en-us/library/aa394582%28v=vs.85%29.aspx