OPEN SPECIFICATION LICENSE (DATED 8/15/02)

1. LICENSE

This Open Specification License (“License”) applies to the Specification Package as described herein.

Any use, reproduction or distribution of any part of the Specification Package constitutes Recipient’s acceptance of this Agreement.

2. DEFINITIONS

Specification Package. The *Specification Package* consists of:

a. The Specification.
b. Reference Implementation.
c. Specification Compatibility Test.
d. Any other related components which may be files, programs and documents related to the form, interface, and semantics of the Specification, or any derivative work.
e. The terms and conditions of this License and any additional licenses that cover specific components considered part of the Specification Package.

Specification. The *Specification* is the document or set of documents as designated by the Original Contributor that defines the form, interface and semantics to the technology covered by the Specification Package.

Reference Implementation. The *Reference Implementation* is a program or set of programs that is designated by the Original Contributor to implement the Specification.

Specification Compatibility Test. The *Specification Compatibility Test* is the program or set of programs that is designated by the Original Contributor to be a fair test of conformance to the Specification.The Original Contributor may, in its sole discretion and from time to time, revise the Specification Compatibility Test

to correct errors and/or omissions and in connection with revisions to the Specifications.

Implementation. An *Implementation* of the Specification is a program or set of programs that defines any class, package or interface whose Program Name is governed by the Name Space designated by the Specification. An *Implementation* is considered conformant if it satisfies the Specification Compatibility Test that is a part of the Specification Package.

Program Name. A *Program Name* is a class, package or interface name. A *Program Name* is governed by a Name Space if the Program Name is a name that starts with the Name Space.

Name Spaces. A *Name Space* is a Program Name, or a specifically designated part of a Program Name. For example, ABCDE.* is the *Name Space* for a collective series of Program Names such as ABCDE.01, ABCDE.BLETCH, etc.

Recipient. *Recipient* means anyone or entity who receives any part of the Specification Package under this Agreement.

Contributor. A *Contributor* is any Recipient that develops, revises or prepares a derivative or change to any part of the Specification Package.

Original Contributor. An *Original Contributor* is the initial originator of the Specification from the California Institute of Technology or Sun Microsystems.

Contribution. A *Contribution* means either: a) the initial Specification Package or b) subsequent changes or additions to the Specification Package made by a Contributor and distributed by Contributor or its agent. Contributions do not include additions to the Specification Package which: a) are separate modules of software distributed in conjunction with the Specification Package under their own license agreement and b) are not derivative works of the Specification Package.

Licensed Patents. *Licensed Patents* means those patent claims licensable by a Contributor which necessarily infringe the use or sale of its Contribution alone or when combined with the Specification Package.

6. GRANT OF RIGHTS

3.1 Subject to the terms of this License, the Original Contributors and Contributor grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Specification Package.

3.2 Whereas the Original Contributor claims the copyright in the Name Spaces identified by the Specification, the Original Contributor grants the Licensee a non-exclusive royalty-free License to use the Name Space provided that any implementation satisfies the Specification Compatibility Test.

3.3 Subject to the terms of this Agreement, each Original Contributor and subsequent Contributors hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any. This patent license shall apply to the combination of the Contribution and the Specification Package if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

3.4 Recipient understands that although each Contributor grants the licenses to Recipient as provided herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other person or entity. Each Contributor disclaims any liability to Recipient for claims brought by any other person or entity based on the infringement of intellectual property rights or otherwise. As a condition to exercising the rights and license granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights required, if any,

such as for third party patent license rights to allow Recipient to distribute any components of third parties in the Specification Package.

7. REQUIREMENTS

4.1 Specification. The copyright to the Specification is owned by the Original Contributor. It may not be modified by Recipient. If Recipient further distribute the Specification either in source form or in printed form, Recipient may only distribute the original Specification verbatim as designated by the Original Contributor. The Original Contributor may from time to time revise the Specification to correct errors or omissions or such other changes by the Original Contributor but shall use best efforts to maintain compatibility.

4.2 Reference Implementation. The Recipient is not permitted to distribute a work derived from the Reference Implementation which: a) defines a Name Space designated in the Specification and b) which does not conform to the Specification Compatibility Test as designated by the Original Contributor.

4.3 Specification Compatibility Test. The Specification Compatibility Test may not be modified for use with this Specification. Any derived work of the Specification Compatibility Test may not be used in lieu of the original Specification Compatibility Test for use in testing conformance.

4.4 If Recipient implements this Specification, then Recipient must insure conformance to the Specification Compatibility Test.

4.5 In the case of any ambiguity of the Specification, the semantics of the Specification shall be defined to be as that in the Reference Implementation and Specification Compatibility Test.

8. WARRANTY

THE SPECIFICATION PACKAGE IS PROVIDED ON AN AS IS BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each

Recipient is solely responsible for determining the appropriateness of using and distributing the Specification Package and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

9. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

10.GOVERNING LAW

This License shall be governed by the laws of the State of California, excluding its conflict of laws rules. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

JSR-38 Software License, Version 1.3
Copyright (c) 2002, California Institute of Technology
ALL RIGHTS RESERVED
U.S. Government Sponsorship acknowledged.

Copyright (c) 2002, Sun Microsystems

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions in source and binary forms with or without modification must retain the above copyright notice, this list of conditions and the disclaimer that follows in the documentation and/or other materials provided with the distribution.
2. The end-user documentation included with the redistribution, if any, must include the following acknowledgment:

   "This product includes software developed by JSR-38 (*http://jsr38.jpl.nasa.gov/*)."

   Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.
3. The names "JSR-38" and "JIFI" are names used to identify this software. "JSR-38" and "JIFI" may not be used to endorse nor promote products derived from this software without prior written permission. For written permission, please contact comments@jsr38.jpl.nasa.gov.
4. Products derived from this software may not be called JSR-38 nor JIFI, nor may "JSR-38" nor "JIFI" appear in their name, without prior written permission of JSR-38.

Disclaimer:

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE CONTRIBUTORS TO JSR-38 BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

User bears all risk related to use, quality and performance of the software.

This software consists of voluntary contributions made by many individuals on behalf of JSR-38. For more information on JSR-38, please see:

<http://jsr38.jpl.nasa.gov/>

This page left blank intentionally.

Application Installation (JSR-38) Specification
Final Edition

29 October 2002

This is the specification of the Application Installation architecture. This architecture is a set of interfaces and classes that support the cross-platform and robust installation of software.

The Open Specification License applies to reference implementations of the JSR-38 specification.

The JSR-38 Software License applies to the Reference Implementation and Test Compatibility Kit provided with the JSR-38 specification.

Application Installation Specification - JSR-38
Version 1.2
Status: Final
Release: October 2002

Copyright 2002 California Institute of Technology
1200 East California Boulevard
Pasadena, Ca 91125 USA

Copyright 2002 Sun Microsystems
4150 Network Circle
Santa Clarita, Ca 95054 USA

All rights reserved.

Preface

This document, *Application Installation (JSR-38) Specification*, defines the Application Installation architecture as a set of interfaces and classes that support the cross-platform and robust installation of software.

Target Audience

The target audiences of this specification are those implementors of installer builders who wish to base their implementation upon a standardized approach to installation.

Readers of this specification document are assumed to be conversant in object oriented programming techniques and Java technology.

Document Description

This document is a general specification for key components of an Installation Builder Application Programming Interface (API). *Key components* are those whose presence and specific implementation are important to produce an installation product that satisfies the goals of this specification.

This document:

- Deliberately avoids any attempt to predetermine installation policy.
- Addresses Technology Compatibility Kit (TCK) requirements only.
- Is not a specific product specification.

The document is not all encompassing; it does not describe all features or the details of all components required to produce an installer builder. Many of the APIs possible in a Reference Implementation (RI) are omitted from the specification so as not to be too restrictive to developers of installer builder products. For example, this document leaves much of the Runtime Installer unspecified. The reference implementation for the Runtime Installer that is provided is meant only to demonstrate the specification components and

associated Graphical User Interface (GUI) Builder Reference Implementation.

Parts of the general services infrastructure are described in this document; others are assumed. Instead, the specific service APIs are described but their implementation is not.

Document Organization

This document is organized into the following chapters:

Chapter 1, Introduction, explains the motivation behind the project.

Chapter 2, Overview, describe the two major components of a possible RI of this specification: the Runtime Installer and Builder applications. The completed Runtime Installer is an instance of the use of the specification API described in Chapters 3 through 5. The Builder uses the APIs to manipulate installation objects and assembles them into the Runtime Installer.

Chapter 3, API Specification: Product Definition, explains how Product Definitions are used to define product components and their relationships and describes the specification Product Definition package.

Chapter 4, API Specification: Product Registry, explains how the Product Registry establishes and tracks product components and describes the specification Product Registry package.

Chapter 5, API Specification: Actions and Rules, explains how actions are used to define and implement installation activities and how rules are used to determine if an action should be executed. The chapter describes the specification Action and Rule packages.

Appendix A, Glossary, defines terms used in this specification.

Feedback

Feedback on this specification is welcome and will be evaluated by the specification team. Feedback may be directed to *comments@jsr38.jpl.nasa.gov*. Every response will be evaluated.

Acknowledgements

The Application Installation API specification is a large effort supported by the members, former and current, of the JSR-38 Expert Group who are acknowledged here:

Hewlett Packard	Paul Larkin
IBM	Bryce Curtis, Mark Edwards
iPlanet	Michael Hein, Gowri Sivaprasad, Padmashree Soundar
Jet Propulsion Laboratory	Rachel Catania, Shirley Cizmar, Marti DeMore, Louis Hirsch, Thom McVittie, Lucille Tanear, Megan Thorsen, Sebastian Van Alphen, Paul Wolgast
SAIC	Greg Frazier
Sun Microsystems	Paul Lovvik, Eric Nielsen, Junaid Saiyed, Michael Wagner, Wenzhou Wang, Matthew Williamson, Kevin Wu, John Xiao
Tripwire	Brad Andersen
Zero G Software	Eric Shapiro

This page left blank intentionally.

Chapter 1 Introduction 1
1.1 Goals 1
1.2 Scope 2
1.3 Guiding Principles 2
1.4 References 3

Chapter 2 Overview 5
2.1 Specification Overview 5
2.2 Reference Implementation Overview 7
2.2.1 Runtime Installer 8
2.2.1.1 Runtime Installer Architecture 9
2.2.1.2 Bus 11
2.2.1.3 Services 12
2.2.1.3.1 Product Definition Service 12
2.2.1.3.2 Product Registry Service 12
2.2.1.3.3 File Service 12
2.2.1.3.4 Filesystem Service 12
2.2.1.3.5 Exec Service 13
2.2.1.3.6 Archive Service 13
2.2.1.3.7 Action and Rules 13
2.2.1.3.8 Other Services 13
2.2.2 Builder 13

Chapter 3 API Specification: Product Definition 15
3.1 Overview 15
3.1.1 Product Definition Elements 16
3.1.2 Dependency Relationships 17
3.1.3 Installation and Registration 18
3.1.4 Uninstallation and Unregistration 20
3.2 Product Definition Package 22
3.2.1 Description 22
3.3 javax.jifi.system.UniqueID 24
3.4 javax.jifi.progress.ProgressContext 24
3.5 Public Interface Installable 24
3.5.1 Syntax 24
3.5.2 Description 24
3.5.3 Methods 26
3.6 Public Interface Suite 29
3.6.1 Syntax 29

3.6.2 Description 29
3.7 Public Interface Product 29
3.7.1 Syntax 29
3.7.2 Description 29
3.8 Public Interface Feature 30
3.8.1 Syntax 30
3.8.2 Description 30
3.9 Public Interface Component 30
3.9.1 Syntax 30
3.9.2 Description 30
3.10 Public Interface Unit 30
3.10.1 Syntax 30
3.10.2 Description 30
3.10.3 Methods 31
3.11 Public Interface Registerable 31
3.11.1 Syntax 31
3.11.2 Description 31
3.11.3 Methods 32
3.12 Example: Public Class ActionUnit 34
3.12.1 Syntax 34
3.12.2 Description 34
3.12.3 Constructors 36
3.12.4 Methods 36
3.13 Example: Public Class FileSetUnit 39
3.13.1 Syntax 39
3.13.2 Description 39
3.13.3 Constructors 41
3.13.4 Methods 42

Chapter 4 API Specification: Product Registry 47
4.1 Overview 47
4.1.1 Abstract Product Structure 48
4.1.2 Queries 49
4.1.3 Component Identification 50
4.1.3.1 Unique Identifiers 50
4.1.3.1.1 UUID Guidelines 52
4.1.4 Component Description 54
4.1.5 Component Registration 54

4.1.5.1 Multiple Versions of a Product Component 56
4.1.5.2 Multiple Instances of a Component 56
4.1.5.3 Product Registry Clients 56
4.1.6 Component Queries 56
4.1.7 Component Removal 57
4.1.8 Component Updates 58
4.2 Product Registry Behavior 59
4.2.1 Component Installation 59
4.2.2 Updating a Component 60
4.2.3 Removing a Component 60
4.3 Product Registry Interfaces and Classes 61
4.3.1 Public Interface RegistryKey 61
4.3.1.1 Syntax 61
4.3.1.2 Description 61
4.3.1.3 Methods 62
4.3.2 Public Interface RegistryComponent 63
4.3.2.1 Syntax 63
4.3.2.2 Description 63
4.3.2.3 Methods 67
4.3.3 Example: Public Class RegistryQuery 75
4.3.3.1 Syntax 75
4.3.3.2 Description 75
4.3.3.3 Constructors 77
4.3.3.4 Methods 77

Chapter 5 API Specification: Actions and Rules 81
5.1 Actions 81
5.1.1 Public Interface Action 83
5.1.1.1 Syntax 83
5.1.1.2 Description 83
5.1.1.3 Methods 84
5.1.2 Public Interface ActionContainer 86
5.1.2.1 Syntax 86
5.1.2.2 Description 86
5.1.2.3 Methods 87
5.1.3 Example: Public Class ExecAction 88
5.1.3.1 Syntax 88
5.1.3.2 Description 89

5.1.3.3 Constructors 91
5.1.3.4 Methods 91
5.1.4 Example: Public Class AddWinRegKeyAction 97
5.1.4.1 Syntax 97
5.1.4.2 Description 97
5.1.4.3 Constructors 98
5.1.4.4 Methods 99
5.2 Rules 102
5.2.1 Public Interface Rule 103
5.2.1.1 Syntax 103
5.2.1.2 Description 103
5.2.1.3 Methods 103
5.2.2 Public Interface RuleSet 104
5.2.2.1 Syntax 104
5.2.2.2 Description 104
5.2.2.3 Methods 104
5.2.3 Example: Public Class WinRegKeyExists 106
5.2.3.1 Syntax 106
5.2.3.2 Description 106
5.2.3.3 Constructors 106
5.2.3.4 Methods 107
5.3 Describable 107
5.3.1 Public Interface Describable 107
5.3.1.1 Syntax 107
5.3.1.2 Description 107
5.3.1.3 Methods 108

Appendix A Glossary 109

Chapter 1 Introduction

The JSR-38 Expert Group, convened under the Java Community Process, has been tasked with the responsibility to produce a specification for a set of Application Programming Interfaces (APIs) that would enable cross platform installation and de-installation of applications. This section defines the scope and purpose of the Application Installation - JSR-38, a set of interfaces and classes that support the cross-platform and robust installation of software.

1.1 Goals

The creators of this specification have as their goal the creation of a installer builder that avoids the pitfalls that often plague software installation. The goals include the ability to:

- Correctly install an application. By this is meant:
  - Properly detect prerequisites to an application before performing the installation.
  - Properly register component requirements.
  - Correctly measure the required disk space, accounting for already installed prerequisite components.
- Correctly uninstall an application. By this is meant:
  - Properly detect if a target component requires other components of the application to be removed and do not remove the targeted component.
  - If removing a target component that requires other components, remove the requirement references from the required components so they can be properly removed at a later time.

1.2 Scope

The focus of this specification is to create an API that facilitates the creation of an installer builder. This specification focuses on a small subset of the features of an installer builder. Only those components that are key to achieving the goals stated above are specified in this document. There are two reasons why this document focuses on a few key areas for this specification:

1. The area of specification is potentially very large. An Installer Builder API spans a large array of services required to deploy and install software.
2. Commercial developers do not want policy imposed on key areas of the implementation such as the runtime engine of the installer or the persistence mechanism of the builder. Therefore, only those key areas that address the most important requirements are considered for this specification.

1.3 Guiding Principles

The following guiding principles are high-level statements that set the general goals of these requirements. These principles guide both the selection of the areas of specification as well as their implementation.

The Application Installation APIs will be:

- **Flexible.** The APIs will provide sufficient functionality to accommodate any installation contingency.
- **Extensible.** The APIs will be able to accommodate new installation requirements without requiring a rewrite of the API.
- **Portable.** The APIs will be able to support any Java 1.1 or greater based virtual machine. Further, the APIs will be portable to all operating systems that support the Java technology.
- **Robust.** The APIs will be able to integrate independent software products into an integrated whole. System and application resource conflicts will be properly managed through

mechanisms known by installation builders and installers resulting from these APIs.

- **Standards-based.** Wherever possible, the APIs will utilize standard software practices and technologies in its implementation. However, inclusion of other technologies must not jeopardize the royalty-free status of the resulting product.
- **Small.** The APIs must be compact and easily understood. The goal of this specification is to produce core, useful and reusable components and leave the bells and whistles to the installer builder software vendors.

1.4 References

The following web site provides background information relevant to the Application Installer:

http://java.sun.com/products/javabeans/docs/index.html

Contains information on JavaBean architecture and pointers to associated tutorials.

This paragraph left blank intentionally.

Chapter 2 Overview

This chapter contains two overviews: an overview of the Application Installation Specification APIs and an overview of a possible Reference Implementation (RI) that uses the specification APIs. The specification APIs provide the means for manipulating installation objects that are assembled into a Runtime Installer. This chapter describes some of the Installer components that are used to deploy and activate an installation.

2.1 Specification Overview

For a Reference Implementation to be considered compliant with this specification, all required interfaces must be implemented and must pass the Technology Compatibility Kit (TCK). The required interfaces are listed below:

javax.jifi.service.product.
- Installable
- Registerable

javax.jifi.service.productregistry.
- RegistryKey
- RegistryComponent

javax.jifi.system.
- Describable

javax.jifi.action.
- Action
- ActionContainer

javax.jifi.rule.
- Rule
- RuleSet

The following interfaces are used by some of the interfaces above. The interfaces below are not required. They are included here for completeness.

javax.jifi.system.
    UniqueID

javax.jifi.progress.
    ProgressContext

javax.jifi.service.productregistry.
    ComponentType

While use of the following interfaces is not required, it is recommended. These interfaces are described fully in this specification and they are used in the reference implementation.

javax.jifi.service.product.
    Component
    Feature
    Product
    Suite
    Unit

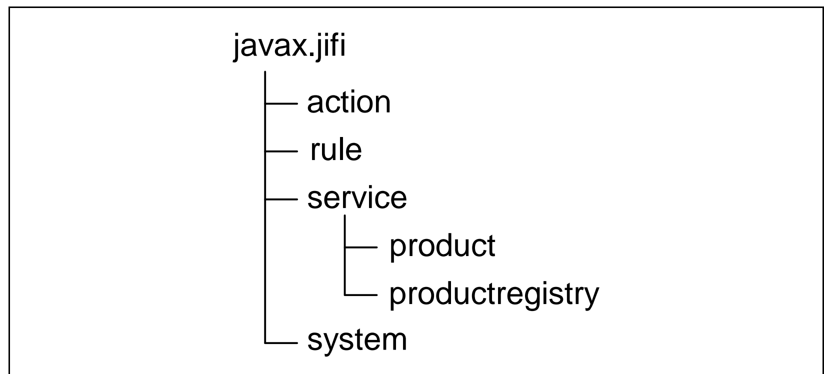
This specification covers the following component interfaces selected based on the goals stated in Chapter 1:

- The Product Definition interface (see Chapter 3) provides a standardized way of organizing the application to install. The application is defined with enough flexibility and extensibility that it can be tailored to match any installation organization. Further, this interface ties directly into the most important of the interfaces in this specification, the Product Registry. This specification defines the Product Definition's relationship with the Product Registry.
- The Product Registry interface (see Chapter 4) addresses one of the most important goals of this specification — that of robustness. The ability to correctly install an application rests primarily on the Product Registry specification and the Product Definition's use of it.

- The Action and Rule interfaces (see Chapter 5) provide a means of plugging behavior into installer implementations without placing requirements on the runtime engine. These two interfaces can provide much of the programmable behavior of an installer without presuming anything about the underlying implementation of the runtime engine.

The API described in this specification addresses the package hierarchy shown in Figure 2-1.

Figure 2-1. Specification Package Hierarchy



It is envisioned that, typically, an implementation of this specification will consist of two major components: a Runtime Installer and Builder application. The Runtime Installer will incorporate the Builder API described in Chapters 3 through 5. A Builder Software Development Kit (SDK) can be built around the specification to provide the ability to develop an Installer Builder application. The framework of such a Builder application would provide a means of customizing and integrating all of the runtime Bean components into a viable Runtime Installer.

2.2 Reference Implementation Overview

Note: The Runtime Installer RI described in this section represents one example of an implementation of a runtime

installer. It is not intended to impose any requirements or standards on how to build the runtime product.

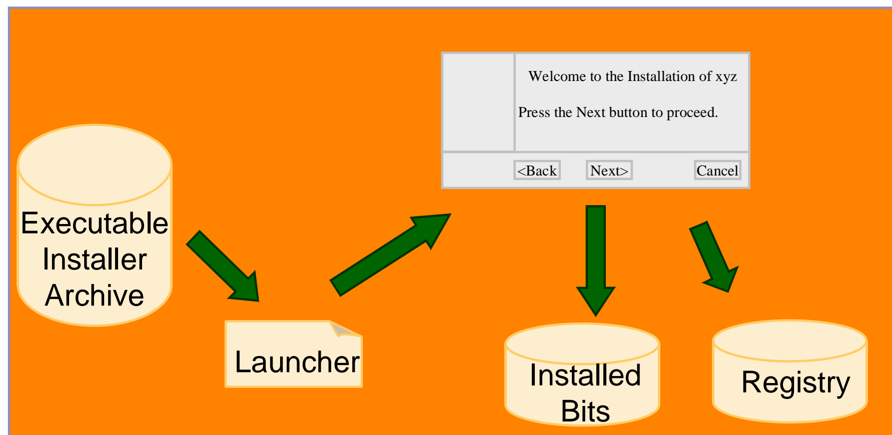
This section:

- Describes the two major components of the RI: the Runtime Installer and Builder application. The completed Runtime Installer is an instance of the use of the Builder API described in Chapters 3 through 5.
- Describes at a high level those aspects of the Builder application used to generate the Runtime Installer application. The Builder provides the APIs for manipulating installation objects that are assembled into a Runtime Installer. The Builder uses a Java Bean component model. The Builder framework provides a means of customizing and integrating all of the runtime Bean components into a viable Runtime Installer.

2.2.1 Runtime Installer

Figure 2-2 illustrates the Runtime Installer context. The figure provides a high level graphic of the Runtime Installer.

Figure 2-2. Runtime Installer Context



The *Executable Installer Archive* is a JAR file that contains the installer executable and all of the installation components used for

a complete application installation. When the JAR file is executed, the entry point is the *Launcher* class, which extracts the Runtime Installer from the JAR and passes control to the runtime. The Runtime Installer presents the user with a series of panels prompting the user for information and allowing forward and backward navigation through the panels. The installation proceeds to install the application files on the target system and updates the *Registry* with the information about the registerable components.

2.2.1.1 Runtime Installer Architecture

Figure 2-3 on page 9 illustrates the modules that make up the Runtime Installer. Table 2-1 on page 10 describes the Runtime Installer modules identified in Figure 3-2.

Note: Only the Product Registry and Product Definition are addressed by this specification.

Figure 2-3. Runtime Installer Modules.

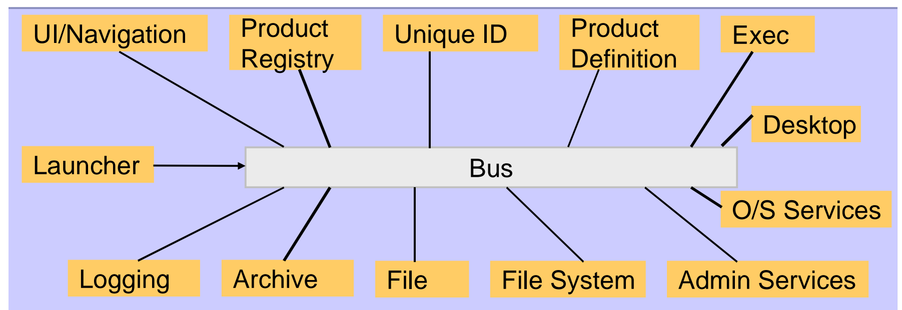


Table 2-1. Runtime Installer Module Descriptions

Module	Description
Admin services	Provides Platform Independent (P/I) services such as user add, etc.
Archive	Provides resource storage and retrieval.
Bus	Provides the communication infrastructure.
Desktop	Provides P/I service to add icons, menus, etc.
Exec	Provides P/I mechanism for running custom executables.
File	Provides platform independent P/I file manipulation service
Filesystem	Provides P/I partition utility for determining information on available file systems, available space and whether they are writable or read only.
Launcher	Provides means to bootstrap the installer from the JAR file
Logging	Provides a publish/subscribe logging mechanism.
O/S Services	Provide P/I services such as reboot, run level, etc.
Product Definition	Contains product and provides services to interrogate, extract and modify.
Product Registry	Provides registration service for all registerable items.

Module	Description
Property	Provides storage and retrieval of properties.
UI/Navigation	Intermediates user input and Runtime Installer.
Unique ID	Provides P/I mechanism for assigning unique identifiers

The Runtime Installer consists of the modules described in Table 2-1.

The implementation provides a *zero footprint installer* solution. That is, the Runtime Installer code resides inside an executable JAR file alongside the application being installed. All other required files — from text files to graphic images — also reside in the same archive.

The Launcher classes provide a means to extract the Runtime Installer from the JAR and pass control to it. Once this is accomplished, the Runtime Installer presents panels to the user and prompts for input. The user answers the questions and presses the “Next” button to proceed to the next panel. This process continues until the installation is complete.

2.2.1.2 Bus

The Bus is a set of interfaces and classes that provide the communication infrastructure for the Runtime Installer. It provides a standard mechanism for communication with and between services. Incorporated into its design is a transparent network capability that allows applications to be installed on a target remote machine without any changes to the service modules. The Bus design also provides a service interface for standardized service implementations. Because the areas that the Bus addresses are installer engine specific, this part of the design has been excluded from the specification.

2.2.1.3 Services

This section will provide a brief overview of the major services the Runtime Installer uses to complete an install.

2.2.1.3.1 Product Definition Service

The Product Definition service provides the information about the application to be installed. It also is responsible for installing the product. Each component of the Product Definition is a customizable Java Bean.

2.2.1.3.2 Product Registry Service

The Product Registry service provides a means of recording the installation of components on the system. It also provides a means of describing dependency relationships between components. The Runtime Installer uses the Product Registry to determine if required components are already on the system, which helps to determine more accurately how much disk space an installation will need. The runtime uninstaller also uses the Product Registry to correctly remove components based upon relationships with other components.

2.2.1.3.3 File Service

The File service provides a platform-independent means of manipulating files on disk. This service is used by the Product Definition service to install files on the system. The minimum level of support is described by the java.io.File class. However, platform specific capabilities are provided for Unix and Windows variants that extend the capability of this service.

2.2.1.3.4 Filesystem Service

The Filesystem service provides information about the availability and size of file systems on the target machine. The installer uses this service to determine where to install the application, how much space is available, etc.

2.2.1.3.5 Exec Service

The Exec service provides a means of executing custom procedures outside the Runtime Installer. The Exec service provides users with an ability to execute any application or script to complete an installation. This facility provides great flexibility that is often required to meet complex installation requirements.

2.2.1.3.6 Archive Service

The Archive service provides a means of organizing and storing all of the files required to complete an installation. The Launcher and Runtime Installer extract all required files using this service. Additionally, the class loader finds all class files needed to execute the Runtime Installer and Launcher in the archive.

2.2.1.3.7 Action and Rules

The Action and Rule classes provide one of the major points of installer customization. Each Action and Rule is included in the Builder as a Java Bean. Because these interfaces describe a major point of customization, these interfaces are part of the required specification API.

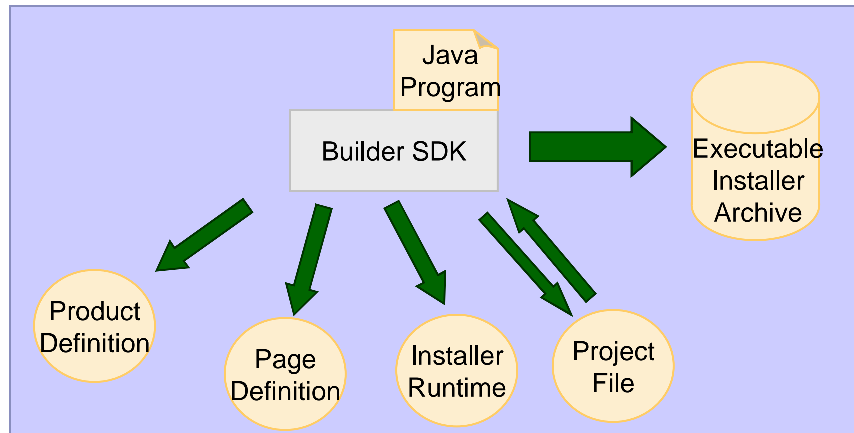
2.2.1.3.8 Other Services

The remaining services not covered here are described in greater detail in their respective section of the RI Javadoc.

2.2.2 Builder

Figure 2-4 on page 14 illustrates the Builder context. The figure provides a high level graphic of the Builder.

Figure 2-4. Builder Context



This diagram describes a possible Builder implementation using the Reference Implementation of the Builder SDK where the Java Program is the Builder application. The Builder application:

- Constructs the Product Definition using the modules described in Chapter 3. The Product Definition describes the product to be installed.
- Defines and constructs the Page Definition (not described in this specification). The Page Definition contains the description and ordering of the graphical panels presented to the user during the installation.
- Generates the Runtime Installer, which contains all the classes needed to execute the installer. All of these components are assembled into the Installer Archive, in this case a JAR file.

All of the user supplied information required to produce this highly customized installer is finally saved in the Project File through a persistence mechanism, such as Java native serialization. This allows the Builder application to reload the customization settings at a later time.

Chapter 3 API Specification: Product Definition

The Product Definition package specification API is described in this chapter. Not all APIs are described; only those required by the specification (see Chapter 2).

For the detailed, definitive description of the APIs, including complete field and method explanations, refer to the Javadoc generated from the specification source distribution.

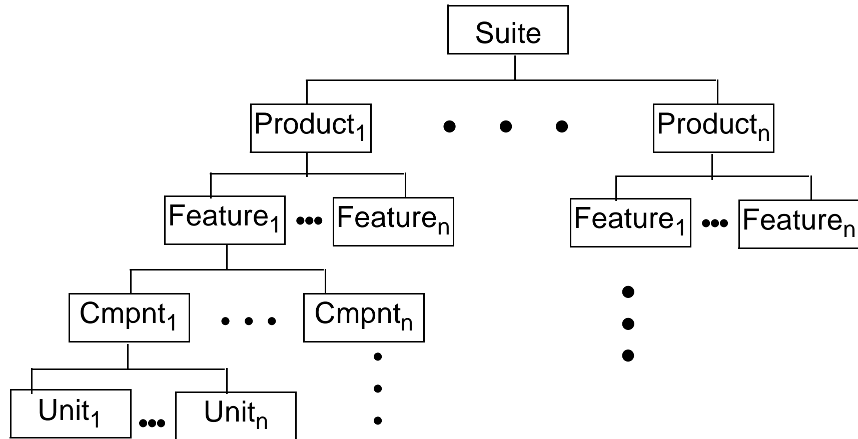
3.1 Overview

A *Product Definition* is a hierarchical data structure describing the components of a product and their relationships to one another. The Product Definition provides:

- A coherent structure for organizing a product makeup.
- The means to install the product described by the Product Definition.
- A means to uninstall an installed component.
- The ability to express dependency relationships between Product Definition components.
- A means, through the use of the Product Registry API, to register all of the components in the Product Definition.

The Product Definition describes the organization of the installables of an installer system. Installables are divided into five hierarchical levels -- suite (highest), product, feature and component, and unit (lowest). Figure 3-1 on page 16 illustrates the hierarchical relationships of installer organizations.

Figure 3-1 Product Definition Hierarchy



Every installer built with the Builder described in this specification contains a Product Definition. The Java classes used in the Product Definition are all derived from a few *superclasses*, which contain code common to all objects within the product tree. These superclasses contain the code that transparently makes use of the install registry, using information such as a product component's Universal Unique Identifier (UUID) and version, that is supplied by the creator of the product tree. A UUID uniquely tags each product component. A component in the Product Definition is *registerable* if it has a UUID, version, and install location. (For more information on UUIDs, see "Unique Identifiers" on page 50.)

3.1.1 Product Definition Elements

Figure 3-1 illustrates the recommended hierarchy for software distribution with the JSR-38 Application Installation framework. Other hierarchies are possible. The underlying interfaces that support the tree do not specify whether a component is a Suite, Product, Feature, Component or Unit. Rather, each component

has zero or more children and a reference to its parent (if there is one).

Suites represent the top of a composite Product Definition. Not all Product Definitions contain the Suite level. Suites are used to aggregate Products.

Products represent the top of a complete installation. Every Product Definition must have at least one Product.

Features represent the smallest end-user selectable unit of software. Features consist of one or more Components. Every Product must have at least one Feature.

Product Components represent the smallest registerable class of the Product Definition. Product Components are contained in Features. Each Feature must have at least a single Product Component. Product Components contain Installable Units.

Installable Units represent the bottom of the Product Definition hierarchy. The term Installable unit is a generic term to describe any type of unit that is assigned to a Product Component.

This chapter also includes the FileSetUnit and the ActionUnit classes as examples of Unit implementations. The FileSetUnit class provides a collection of files that can be installed as a single unit. This unit can also be parented by a Product Component and thus registered as one unit. The ActionUnit class provides a means of packaging an action into a registerable component and placing it in the Product Definition. The install() method calls the underlying Action's execute() method. (See Chapter 5.)

3.1.2 Dependency Relationships

The Product Definition is required to register dependency relationships through the use of the Product Registry API. The relationship of one component as dependent on another component is required. Also the corollary relationship must be recorded, that of one component requiring another component.

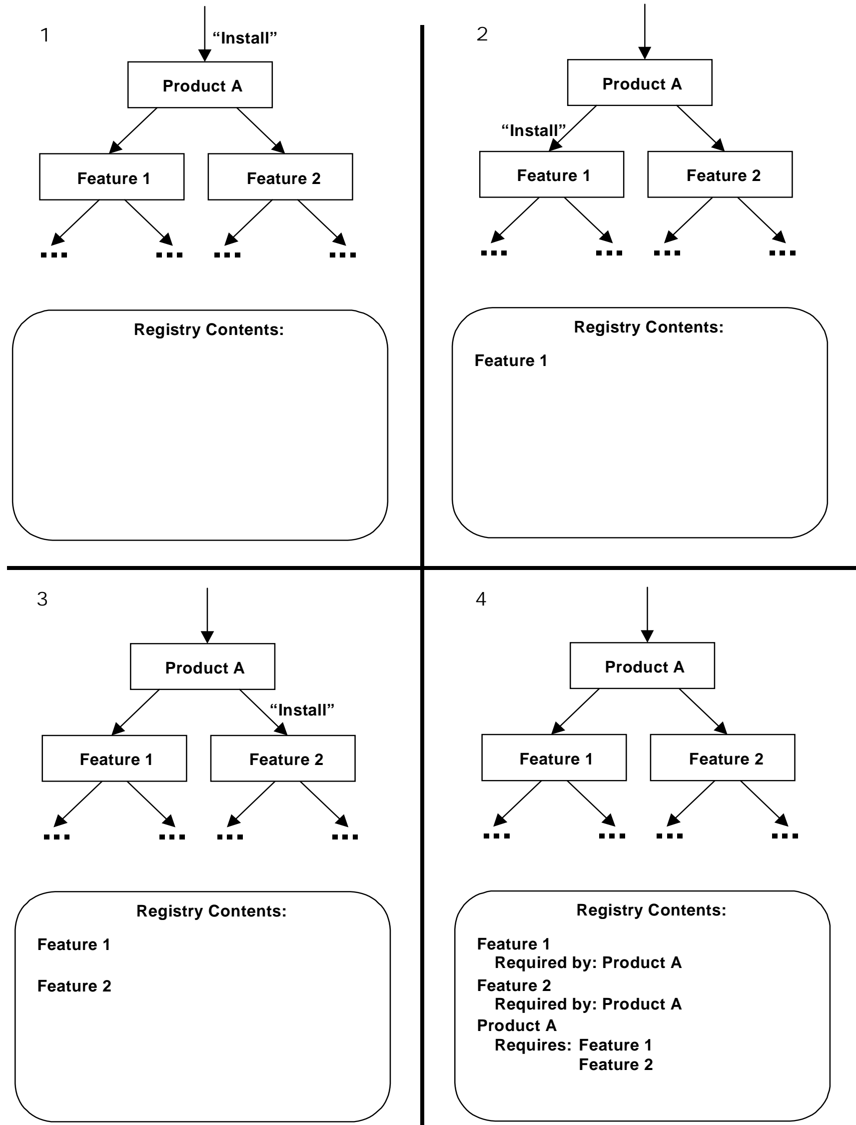
3.1.3 Installation and Registration

Each object in the product tree has one parent and zero or more children except for the root component, which has no parent. During installation, the product tree is traversed depth-first, and each node encountered *installs* itself according to a recursive set of rules.

If the node is not registerable, it is simply installed normally, and no registry interaction takes place.

If the node is registerable, it first queries the registry. (See Figure 3-2, Frame1.) If the component is already installed, nothing happens. Otherwise, it must first install all of its children. (See Figure 3-2, Frames 2 and 3.) Then, it installs and registers itself and registers the fact that it is dependent on each of its registerable children that just installed themselves prior to this. (See Figure 3-2, Frame 4.)

Figure 3-2 Installation Process



Note that:

- Each level can contain one or more of the next lower level of installables. For example, a suite consists of one or more products; a product consists of one or more features, and so on.
- The lowest installable level is the unit. The FileSetUnit is an example of a set of files that are grouped together and implemented as a unit. They are prepared as a unit and parented with a component. When they are installed, they are installed as a unit. Note that while all other installable levels can be registered, units cannot.

3.1.4 Uninstallation and Unregistration

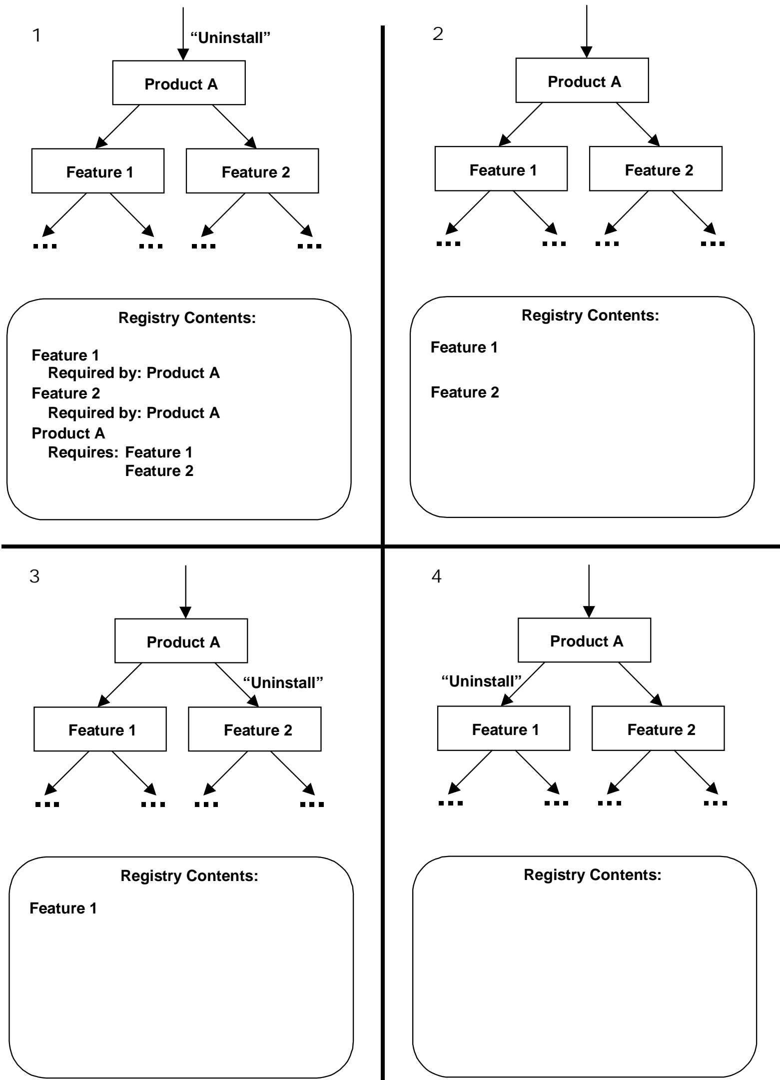
The process of uninstallation is the exact reverse of installation.

During uninstallation, the product tree is traversed depth-first, and each node encountered uninstalls itself according to a recursive set of rules:

If the node is not registerable, it is simply uninstalled normally, and no registry interaction takes place.

If the node is registerable, it first queries the registry. (See Figure 3-3, Frame 1.) If there are components that depend on this component, nothing happens. Otherwise, the component first unregisters itself completely, including the dependencies on its children that are registered. (See Figure 3-3, Frame 2.) It then physically uninstalls itself from the system. Finally, it uninstalls each of its children in reverse order. (See Figure 3-3, Frames 3 and 4.)

Figure 3-3 Uninstallation Process



3.2 Product Definition Package

This specification describes only those interfaces and classes of the Product Definition required to provide a robust installation. The remainder of the API can be considered artifacts of the Reference Implementation (RI). The required part of the Product Definition are the five levels of installables. Each of these extend the Installable interface which provides the functionality for each component in the Product Definition.

Note: The Product Query capability is implementation dependent and is, therefore, not discussed in the specification.

3.2.1 Description

Each of the seven main Product Definition interfaces describes:

- A method called install (), responsible for installing the component to which it is attached.
- A method called uninstall(), responsible for uninstalling the component to which it is attached.

Class Summary	
Interfaces	
`Component`	Identifies a bean in a Product Definition hierarchy. A component is between a Feature and a Unit and is the lowest registerable level of a Product Definition.
`Feature`	Identifies a bean in a Product Definition. A Feature is between a Product and a Component in a Product Definition hierarchy.

Class Summary	
`Installable`	This interface must be implemented by all classes in the product definition that are installed.
`Product`	Identifies a bean in a Product Definition. A Product falls between a Suite and a Feature in a Product Definition hierarchy.
`Registerable`	This interface must be implemented by all objects in the Product Definition that are to be registered.
`Suite`	Identifies a bean in a Product Definition. A Suite is an optional top-level object in a Product Definition hierarchy.
`Unit`	Identifies a bean in a Product Definition. A Unit is the lowest element in a Product Definition that is responsible for modifying the state of the target machine during an installation.
Classes	
`ActionUnit`	A cross-platform installation unit that executes Actions.
`FileSetUnit`	A cross-platform installation unit that installs files.

3.3 javax.jifi.system.UniqueID

The UniqueID class, used throughout the Product Definition, is expected to be in the javax.jifi.system package (not described in this document). The definition of the UniqueID class is implementation specific. A more complete discussion of Unique Identifiers is presented in the section “Unique Identifiers” on page 50.

3.4 javax.jifi.progress.ProgressContext

The ProgressContext class, used throughout much of this document, is expected to be in the javax.jifi.progress package (not described in this document). The definition of the ProgressContext class is implementation specific and optional. This class reports the progress of operations that take a long time to perform. Use of this class provides a means to give feedback to the user via a GUI front end on the state of the application.

3.5 Public Interface Installable

3.5.1 Syntax

```
public interface Installable extends java.io.Serializable
```

3.5.2 Description

This interface must be implemented by all classes in the Product Definition that are installed. The Product Definition is essentially a description of what is to be installed and what rules apply to that installation. The Product Definition is also used to accomplish uninstall.

Member Summary	
Methods	
`getID()`	Returns the unique id that is used to identify this Installable.

Member Summary	
`getInstallLocation()`	Returns the destination directory for this product bean.
`getInstallRule()`	Returns the rule that is used to determine whether or not this Installable should install.
`getUninstallRule()`	Returns the rule that is used to determine whether or not this Installable should uninstall.
`getVersion()`	Returns the version string that is used to identify this Installable.
`install()`	Installs the bits associated with this Installable.
`isInstallable()`	Returns true if this product bean is installable according to evaluation of the install rule; false otherwise.
`isUninstallable()`	Returns true if this product bean is uninstallable according to evaluation of the uninstall rule; false otherwise.
`setID(UniqueID)`	Sets the id that is used to uniquely identify this Installable.
`setInstallLocation (String)`	Sets the destination directory for this product bean.
`setInstallRule(Rule)`	Sets the rule that determines whether or not this Installable should install.
`setUninstallRule(Rule)`	Sets the rule that determines whether or not this Installable should uninstall.

Member Summary	
`setVersion(String)`	Sets the version string that is used to identify this Installable.
`uninstall()`	Uninstalls the bits associated with this Installable.

3.5.3 Methods

getID()

```
public UniqueID getID()
```

Returns the unique id that is used to identify this Installable.

Returns: The UUID used to identify this Installable.

getInstallLocation()

```
public java.lang.String getInstallLocation()
```

Returns the destination directory for this product bean. If the install location is not set, may return null.

Returns: The destination directory.

getInstallRule()

```
public Rule getInstallRule()
```

Returns the rule that is used to determine whether or not this Installable should install. If the rule is not set, may return null.

Returns: The install rule.

getUninstallRule()

```
public Rule getUninstallRule()
```

Returns the rule that is used to determine whether or not this Installable should uninstall. If the rule is not set, may return null.

Returns: The uninstall rule.

getVersion()

```
public java.lang.String getVersion()
```

Returns the version string that is used to identify this Installable. If the rule is not set, may return null.

Returns: The String representing the version.

install()

```
public void install()
```

Installs the bits associated with this Installable. The installation will only occur if the install rule (if provided) evaluates to true.

isInstallable()

```
public boolean isInstallable()
```

Returns true if this product bean is installable according to evaluation of the install rule; false otherwise. If the install rule is not set, returns true.

isUninstallable()

```
public boolean isUninstallable()
```

Returns true if this product bean is uninstallable according to evaluation of the uninstall rule; false otherwise.

setID(UniqueID)

```
public void setID(UniqueID id)
```

Sets the id that is used to uniquely identify this Installable.

Parameters:

`id` - the UUID used to identify this Installable

setInstallLocation(String)

```
public void setInstallLocation(java.lang.String
          installLocation)
```

Sets the destination directory for this product bean.

Parameters:

`installLocation` - the destination directory for this Installable

setInstallRule(Rule)

```
public void setInstallRule(Rule installRule)
```

Sets the rule that determines whether or not this Installable should install.

Parameters:

`installRule` - the rule that evaluates to true only if this Installable should be installed

setUninstallRule(Rule)

```
public void setUninstallRule(Rule uninstallRule)
```

Sets the rule that determines whether or not this Installable should uninstall.

Parameters:

`uninstallRule` - the rule that evaluates to true only if this Installable should be uninstalled

setVersion(String)

```
public void setVersion(java.lang.String version)
```

Sets the version string that is used to identify this Installable.

Parameters:

`version` - the version string

uninstall()

```
public void uninstall()
```

Uninstalls the bits associated with this Installable. The uninstallation will only occur if the uninstall rule (if provided) evaluates to true.

3.6 Public Interface Suite

3.6.1 Syntax

```
public interface Suite extends Installable, Registerable
```

3.6.2 Description

Identifies a bean in the Product Definition hierarchy. A suite is an optional top-level object in the hierarchy. A suite contains products, which contain features, etc.

3.7 Public Interface Product

3.7.1 Syntax

```
public interface Product extends Installable, Registerable
```

3.7.2 Description

Identifies a bean in a Product Definition. A `Product` falls between a `Suite` and a `Feature` in the Product Definition hierarchy. A `Product` is registerable.

3.8 Public Interface Feature

3.8.1 Syntax

```
public interface Feature extends Installable, Registerable
```

3.8.2 Description

Identifies a bean in a Product Definition. A `Feature` is between a `Product` and a `Component` in the Product Definition hierarchy, and generally represents optional parts of a product. A `Feature` is registerable.

3.9 Public Interface Component

3.9.1 Syntax

```
public interface Component extends Installable,
   Registerable
```

3.9.2 Description

Identifies a bean in a Product Definition hierarchy. A component is between a `Feature` and a `Unit`, and is the lowest level of the Product Definition that is registerable.

3.10 Public Interface Unit

3.10.1 Syntax

```
public interface Unit extends Installable
```

3.10.2 Description

Identifies a bean in a Product Definition. A `Unit` is the lowest level of a Product Definition that is responsible for modifying the state of the target machine during an installation. A `Unit` is not registerable.

Member Summary	
Methods	
`isEmpty()`	Returns a flag indicating whether or not a unit contains installable items.

3.10.3 Methods

isEmpty()

```
public boolean isEmpty()
```

Returns a flag indicating whether or not a unit contains installable items. Used by verification and to determine if the unit can be cleaned up (removed).

3.11 Public Interface Registerable

3.11.1 Syntax

```
public interface Registerable
```

3.11.2 Description

This interface must be implemented by all objects in the Product Definition that are to be registered. The lowest valid object that can implement this interface is Component.

Member Summary	
Methods	
`getComponentDescription()`	Gets the RegistryComponent object for this registerable.

Member Summary	
`getRegisteredComponent Description()`	Gets the RegistryComponent object of this registerable as put in the registry.
`getToBeRegistered()`	Retrieves a flag indicating whether the component is to be registered in the Product Registry during installation.
`isRegistered()`	Returns true if this registerable has already been registered.
`register()`	Registers this registerable.
`setComponentDescription (RegistryComponent)`	Sets the RegistryComponent object for this registerable.
`setToBeRegistered(boolean)`	Sets the flag indicating whether the component is to be registered in the Product Registry during installation.
`unregister()`	Unregisters this registerable.

3.11.3 Methods

getComponentDescription()

```
public RegistryComponent getComponentDescription()
```

Gets the RegistryComponent object for this registerable. If the component description is not set on the Registerable, may return null.

Returns: The RegistryComponent for this registerable.

getRegisteredComponentDescription()

```
public RegistryComponent
            getRegisteredComponentDescription()
```

Gets the RegistryComponent object for this registerable. If the Registry does not contain this component, may return null.

Implementation of this method may throw a Runtime exception if the underlying service is unavailable until an implementation-specific initialization has been performed.

Returns: The RegistryComponent for this registerable.

getToBeRegistered()

```
public boolean getToBeRegistered()
```

Retrieves a flag indicating whether the component is to be registered in the Product Registry during installation.

Returns: True if the object should be registered; false otherwise.

isRegistered()

```
public boolean isRegistered()
```

Returns true if this registerable has already been registered.

Returns: If this registerable is registered or not.

register()

```
public void register()
```

Registers this registerable.

setComponentDescription(RegistryComponent)

```
public void
            setComponentDescription(RegistryComponent
            componentDescription)
```

Sets the RegistryComponent object for this registerable.

Parameters:

`componentDescription` - the component description for this registerable

setToBeRegistered(boolean)

```
public void setToBeRegistered(boolean register)
```

Sets the flag indicating whether the component is to be registered in the Product Registry during installation.

Parameters:

`register` - true if the object should be registered; false otherwise.

unregister()

```
public void unregister()
```

3.12 Example: Public Class ActionUnit

3.12.1 Syntax

```
public class ActionUnit extends product.UnitBean
```

```
java.lang.Object
  |
  +--product.InstallableBean
        |
        +--product.UnitBean
             |
             +--product.ActionUnit
```

3.12.2 Description

A cross-platform installation unit that executes Actions based upon the Platform rule. This Unit has two actions associated with it, an install action and an uninstall action.

Member Summary	
Constructors	
`ActionUnit()`	No arg constructor.
`ActionUnit(Action, Action)`	Constructor that is initialized with both the install action and the uninstall action.
`ActionUnit (String, UniqueID, Action, Action)`	Instantiates a new ActionUnit with the specified label and UUID that will execute the specified action.
Methods	
`getInstallAction()`	Returns the install action that will be executed at install time.
`getUninstallAction()`	Returns the action that will be executed at uninstall time.
`initializeInstall (ProgressContext)`	Initializes the ProgressContext object provided.
`initializeUninstall (ProgressContext)`	Initializes the ProgressContext object provided.
`install (ProgressContext)`	Executes the install action.
`isEmpty()`	Returns true or false depending on whether this unit contains an action.
`setInstallAction (Action)`	Sets the action that will be executed at install time by this unit.
`setUninstallAction (Action)`	Sets the action that will be executed at uninstall time by this unit.
`uninstall (ProgressContext)`	Execute the uninstall action.
`verify()`	Verifies the integrity of this unit.

3.12.3 Constructors

ActionUnit()

```
public  ActionUnit()
```

No arg constructor.

ActionUnit(Action, Action)

```
public  ActionUnit(Action installAction,
            Action uninstallAction)
```

Constructor that is initialized with both the install action and the uninstall action.

ActionUnit(String, UniqueID, Action, Action)

```
public  ActionUnit(java.lang.String label, Action
            uuid,
            Action installAction, Action
            unInstallAction)
```

Instantiates a new ActionUnit with the specified label and UUID that will execute the specified action.

Parameters:

`label` - the unit's label

`uuid` - the UniqueID for this unit

`archiveFilename` - the file within the installer archive to be installed.

3.12.4 Methods

getInstallAction()

```
public Action getInstallAction()
```

Returns the install action that will be executed at install time.

getUninstallAction()

```
public Action getUninstallAction()
```

Returns the action that will be executed at uninstall time.

initializeInstall(ProgressContext)

```
public ProgressContext initializeInstall
          (progress.ProgressContext progress)
```

Initializes the ProgressContext object provided.

Overrides: `InstallableBean.initializeInstall(ProgressContext)` in class `InstallableBean`

Parameters:
`progress` - the ProgressContext object to initialize

Returns: the ProgressContext object that was initialized

initializeUninstall(ProgressContext)

```
public ProgressContext initializeUninstall
          (ProgressContext progress)
```

Initializes the ProgressContext object provided.

Overrides: `InstallableBean.initializeUninstall(ProgressContext)` in class `InstallableBean`

install(ProgressContext)

```
public ProgressContext install(ProgressContext
          progress)
```

Executes the install action.

isEmpty()

```
public boolean isEmpty()
```

Returns true or false depending on whether this unit contains an action.

Returns: always returns false

setInstallAction(Action)

```
public void setInstallAction(Action action)
```

Sets the action that will be executed at install time by this unit.

Parameters:
`action` - the action to execute at install time

setUninstallAction(Action)

```
public void setUninstallAction(Action action)
```

Sets the action that will be executed at uninstall time by this unit.

Parameters:
`action` - the action to execute at uninstall time

uninstall(ProgressContext)

```
public ProgressContext uninstall(ProgressContext
          progress)
```

Execute the uninstall action.

verify()

```
public verification.UserMessage[] verify()
```

Verifies the integrity of this unit.

Overrides: InstallableBean.verify () in class InstallableBean

Returns: The array of user messages describing the reasons for the verification failure

3.13 Example: Public Class FileSetUnit

3.13.1 Syntax

```
public class FileSetUnit extends product.UnitBean

java.lang.Object
  |
  +--product.InstallableBean
        |
        +--product.UnitBean
              |
              +--product.FileSetUnit
```

3.13.2 Description

A cross-platform installation unit that installs files. Timestamps and permissions are preserved.

Member Summary	
Constructors	
`FileSetUnit()`	No arg constructor.
`FileSetUnit(String, UniqueID, String)`	Instantiates a new FileSetUnit with the specified label and UUID that will install the specified file from the installer archive.
`FileSetUnit(String, UniqueID, String, String)`	Instantiates a new FileSetUnit with the specified label, UUID, and version that will install the specified file from the installer archive.

Member Summary	
Methods	
addProvider(Provider)	Sets the FileSetProvider that will write the data into the install archive.
fillSizeTable (ProgressContext, FilesystemTable)	Fills up the filesystem table with the size (required disk space) for this Installable
getArchiveFilename()	Returns the name of the file that will be installed.
getInstallSize()	Returns the size of this unit.
getRawSize()	Returns the size of this unit.
initializeFillSizeTable (ProgressContext)	Initializes the progress for the recheckSizeTable method
initializeInstall (ProgressContext)	Initializes the ProgressContext object provided.
initializeRecheckSizeTable (ProgressContext)	Initializes the progress for the recheckSizeTable method
initializeUninstall (ProgressContext)	Initializes the ProgressContext object provided.
install(ProgressContext)	Installs the file.
isEmpty()	Indicates whether this fileset contains any files by looking at its FileProviders.
recheckSizeTable (ProgressContext, FilesystemTable)	Checks the flagged off filesystems for more a more fine grained search

Member Summary	
`setArchiveFilename(String)`	Sets the name of the file that will be installed by this unit.
`uninstall(ProgressContext)`	Uninstalls the files in this unit.
`verify()`	Verifies the integrity of this unit.

3.13.3 Constructors

FileSetUnit()

```
public  FileSetUnit()
```

No arg constructor.

FileSetUnit(String, UniqueID, String)

```
public  FileSetUnit(java.lang.String label, UniqueID
            uuid, java.lang.String archiveFilename)
```

Instantiates a new FileSetUnit with the specified label and UUID that will install the specified file from the installer archive.

Parameters:

`label` - the unit's label

`uuid` - the UniqueID for this unit

`archiveFilename` - the file within the installer archive to be installed.

FileSetUnit(String, UniqueID, String, String)

```
public  FileSetUnit(java.lang.String label,
            UniqueID uuid, java.lang.String version,
            java.lang.String archiveFilename)
```

Instantiates a new FileSetUnit with the specified label, UUID, and version that will install the specified file from the installer archive.

Parameters:

`label` - the unit's label

`uuid` - the UniqueID for this unit

`version` - the version string for this file unit

`archiveFilename` - the file within the installer archive to be installed.

3.13.4 Methods

addProvider(Provider)

```
public void addProvider(Provider provider)
```

Sets the FileSetProvider that will write the data into the install archive.

Overrides: `InstallableBean.addProvider(Provider)` in class `InstallableBean`

Parameters:

`provider` - the FileSetProvider that will write the files into the installer archive

fillSizeTable(ProgressContext, FilesystemTable)

```
public void fillSizeTable(ProgressContext pc,
          FilesystemTable filesystemTable)
```

Fills up the filesystem table with the size (required disk space) for this Installable

Overrides:

`InstallableBean.fillSizeTable(ProgressContext, FilesystemTable)` in class `InstallableBean`

Parameters:

`filesystemTable` - the filesystem table to be updated

getArchiveFilename()

```
public java.lang.String getArchiveFilename()
```

Returns the name of the file that will be installed.

getInstallSize()

```
public InstallFilesystems getInstallSize()
```

Returns the size of this unit.

Overrides: `InstallableBean.getInstallSize()` in class `InstallableBean`

getRawSize()

```
public double getRawSize()
```

Returns the size of this unit.

Overrides: `InstallableBean.getRawSize()` in class `InstallableBean`

initializeFillSizeTable(ProgressContext)

```
public void initializeFillSizeTable
          (ProgressContext pc)
```

Initializes the progress for the recheckSizeTable method

Overrides: `InstallableBean.initializeFillSizeTable(ProgressContext)` in class `InstallableBean`

Parameters:
`pc` - the progress context for the fillSizeTable

Returns: the initialize progress context

initializeInstall(ProgressContext)

```
public ProgressContext initializeInstall
          (ProgressContext progress)
```

Initializes the ProgressContext object provided.

Overrides: `InstallableBean.initializeInstall (ProgressContext)` in class `InstallableBean`

initializeRecheckSizeTable(ProgressContext)

```
public void initializeRecheckSizeTable
          (ProgressContext pc)
```

Initializes the progress for the recheckSizeTable method

Overrides:
`InstallableBean.initializeRecheckSizeTable (ProgressContext)` in class `InstallableBean`

Parameters:
`pc` - the progress context for the fillSizeTable

Returns: The initialize progress context.

initializeUninstall(ProgressContext)

```
public ProgressContext initializeUninstall
          (ProgressContext progress)
```

Initializes the ProgressContext object provided.

Overrides: `InstallableBean.initializeUninstall (ProgressContext)` in class `InstallableBean`

install(ProgressContext)

```
public ProgressContext install(ProgressContext
          progress)
```

Installs the file.

isEmpty()

```
public boolean isEmpty()
```

Indicates whether this fileset contains any files by looking at its FileProviders.

Returns: True if no files are included in this FileSetUnit; false otherwise.

recheckSizeTable(ProgressContext, FilesystemTable)

```
public void recheckSizeTable(ProgressContext pc,
          FilesystemTable filesystemTable)
```

Checks the flagged off filesystems for more a more fine grained search

Overrides: `InstallableBean.recheckSizeTable(ProgressContext, FilesystemTable)` in class `InstallableBean`

Parameters:
`filesystemTable` - the filesystem table to be checked

Returns: If the rechecking failed.

setArchiveFilename(String)

```
public void setArchiveFilename(java.lang.String
          archiveFilename)
```

Sets the name of the file that will be installed by this unit.

Parameters:
`archiveFilename` - the name of the file within the installer archive that will be installed

uninstall(ProgressContext)

```
public ProgressContext uninstall(ProgressContext
          progress)
```

Uninstalls the files in this unit.

verify()

```
public verification.UserMessage[] verify()
```

Verifies the integrity of this unit.

Overrides: `InstallableBean.verify()` in class `InstallableBean`

This page left blank intentionally.

Chapter 4 API Specification: Product Registry

The Product Registry (PR) package API is described in this chapter. Not all APIs are described; only those required in the specification.

Each section in this chapter describes a package interface, class, or exception of the API in terms of functionality and use.

For the detailed, definitive description of the API, including complete field and method explanations, refer to the Javadoc generated from the source distribution.

4.1 Overview

The PR is the cornerstone upon which a robust installation is predicated. Without the registry, the ability to manage dependencies, to determine true disk space requirements, and to correctly remove applications would not be possible.

The PR is fashioned like the other services in this specification. The underlying mechanism for implementing the registry is left open and, in fact, two reference implementations of the PR are available, one as a Java implementation, usable on any system, the other as a native implementation available only on Solaris.

The PR provides a means to establish and track installed components and the dependencies installed components have with other components. The purpose of the Product Registry is to provide the mechanism for recording software products and components that are installed.

Installation of software on to robust operating systems involves more than copying bits on to the file system. Reliable product deployment is key to reducing support costs associated with software installation and upgrade. Installation software must be able to detect installation problems and report those problems to the user. This feedback is the key to giving a customer the

necessary confidence in the installation process which results in higher customer satisfaction and lower product support costs.

Products are becoming more sophisticated and often benefit from high-level services provided by the operating system or third-party vendors. These services are generally accessible through a component layer. Simple implementations of component-like services involve linking with shared libraries. More robust component infrastructures require a runtime lookup mechanism, which permits multiple vendor solutions for any particular service. The components are useful outside of any particular product that requires them and components may be deployed independently of any end-user product. As products become more componentized, the problem of software installation and reliable error detection is compounded.

4.1.1 Abstract Product Structure

The Product Definition structuring capabilities are helpful for organizing installation operations including grouping components that require similar handling. A product can be broken up into pieces, some of which may not need to be installed for the product to function. A full product description includes information about the relationship between the product and its features (the pieces of the product that a user can optionally install), and also the relationship between each feature and the individual components used to install that feature.

Note that the model of installed software presented here — *Product* contains *Features* which contain *Components* — is one possible model. Another could be *Load* contains *Base* and *Extended* groupings, which in turn contain *Documentation*, *Executable* and *Source* groupings, which in turn contain *Components* that consist of collections of file system objects to install. In any case, the RegistryComponent object described in the section “Public Interface RegistryComponent” on page 63 can accommodate many containment relationships since each component contains a parent and an array of RegistryKeys for the component’s children.

One responsibility of the Product Registry is to help provide information facilitating the removal of products. Therefore, the relationships between the different pieces of a product must be recorded. If a user tries to remove a product, the uninstall software must be able to determine whether its children have been installed. In the *Product-Feature-Component* model described above, it would be necessary to determine:

- That the product is installed.
- The features of the product that were installed.
- The components that comprise the features of the product.

Dependencies between components must be recorded so that acceptable behavior for removal and upgrade can be enforced.

When one product is dependent on another product, that dependency must be expressed and recorded in such a way that the required component will not be inadvertently removed, resulting in the dependent component being damaged or corrupted.

The Product Registry must be able to store this abstract product structure and be able to serve this information to the clients that request it.

4.1.2 Queries

The Product Registry must support a robust set of product queries to be useful to installation and uninstallation software. The ability to add product information to the registry and remove product information from the registry is primarily provided to maintain the registry state.

The whole point of collecting, recording, and maintaining the product information is to facilitate queries on that data, so reasonable decisions can be made.

A client of the Product Registry should be able to determine if a specific component is installed, given only the component's identifier or unique name. This query will return NULL if no component of that ID is currently installed. If the specified

component is installed, this query will return the latest instance of the latest version of the specified component installed on the system.

A client should also be able to narrow the component search by specifying the component's version or the location in which the component is installed.

Each query results in either a Component Description class that completely describes the specified component if the component is registered or NULL if the specified component is not registered.

4.1.3 Component Identification

The identification of the component is determined by three attributes:

- The component ID, a universally unique identifier for the component.
- The component version, a string that provides information about the component that uniquely identifies the version.
- The unique name, a human readable name that identifies the component.

When registering a component, none of these attributes can be null.

(See "Public Interface RegistryKey" on page 61 for specifications on the component identification interface.)

4.1.3.1 Unique Identifiers

It is very important to be able to uniquely identify product components. A naming conflict between two product components could result in erroneous installation behavior or corruption of the install registry. For example, Solaris products are described in terms of SVR4 packages, each of which is given a package name. If the same name is applied to two separate packages comprised of different contents, the packaging system will assume that the two packages represent the same installable component. This results in confusion of the packaging system

because upon installation of the second such package, the packaging system believes that the package has already been installed.

For example, at Sun Microsystems, the package name is an abbreviated description or name of the package contents, prepended with the company's stock symbol (SUNW). Each package that is released from Sun must have its package name registered on a company registry. If the requested package name has already been granted, a different name must be used. In this way package conflicts between two packages created by Sun can never occur.

This scheme does not guarantee that two packages created by separate companies will not conflict. In order to guarantee that no conflicts will ever arise, there must be a central package name registry which everyone creating an SVR4 package must use.

UUID stands for "Universal Unique Identifier". The idea behind a UUID is that anyone can generate a unique identifier on any computer at any time, and be guaranteed that the identifier will be unique.

If no centralized name registrar exists to coordinate the issuing of unique names, UUIDs function as unique identifiers. There are cases where structured unique identifiers are more appropriate than centrally administered opaque strings such as UUIDs. Therefore, there is no requirement to use UUIDs as unique identifiers for registry components, though it is recommended.

A UUID is unique across both space and time, with respect to all UUIDs. In short, this is accomplished by combining the ethernet address of the machine creating the UUID with the creation time of the UUID. If generated according to the specification, the UUID is either guaranteed to be different from all other UUIDs generated until approximately 3400 AD or extremely likely to be different.

When applied to installation, the UUID makes it possible to easily create identifiers for each product component without repercussions resulting from ID collisions.

Queries for components in the Product Registry can be performed from either the unique name or the component ID. The preferred practice is to query using the component ID. The unique name queries are provided to support package creators used to working with package names.

The product tree in the Web Start Wizard SDK performs all of its queries based on the component ID. The SDK was designed to be platform independent, and component naming conflicts are a hazard that is best avoided.

4.1.3.1.1 UUID Guidelines

This section examines the life cycle of a hypothetical component and maps the events in that life cycle with UUID creation. This is done to illustrate when a UUID should be created/recreated in the component's life cycle.

A component may undergo several events in its life cycle. These events are listed in the first column of Table 4.

Table 4. UUID Creation Mapped to the Component Life Cycle

Component Event	Create UUID?	Effect
Creation of a new component	Yes	The UUID will be used to uniquely identify the new component. It should only be created one; not each time the component is prepared for release (i.e., each time the package is created).

Component Event	Create UUID?	Effect
Creation of a new version of an existing component that maintains compatibility to prior versions of the same component	No	Use the same UUID that was created when the component was created. The UUID identifies the component, not the version of a component.
Transfer of a component to a different organization that results in the name of the component being changed, but the contents and purpose of the component remain the same.	No	Again, the same UUID should be used. The UUID is not meant to identify the owner of the component; not just the component itself.
The component's name changes	No	The UUID insulates a product component from the name of its required components.
The component is broken apart and its pieces (intellectual property) are used to create a different component that has a different purpose than the original one.	Yes	This would be considered a new component. To differentiate this component from the original one, a new UUID should be created.

As the table shows, a UUID should not be generated every time, say, the component's SVR4 package is recreated. The same UUID is used throughout the life of the component.

4.1.4 Component Description

This section details how product components are described for recording in the Product Registry.

The Component Description class is used to describe a product or piece of a product for use in the Product Registry. This structure is written into the registry's data file and describes a particular component. All successful product and component queries result in a Component Description class being copied from the data file and returned to the caller.

The Component Description class should answer any reasonable question about the component that installation or removal software would ask. This includes:

- The component's id
- The component's unique name
- The component's display name
- The component's version
- The component's parent
- The install location
- The vendor of the component
- The list of components this component requires
- The list of components required by this component
- The list of child components
- A list of versions this component is backward compatible with
- Other application-specific data can be included in the structure in the form of key/value pairs.

(See "Public Interface RegistryComponent" on page 63 for specifications on the component description interface.)

4.1.5 Component Registration

Registration of a component is the act of adding a Component Description class to the data file. Registering components from a client's point of view is accomplished using the following steps:

1. Create an instance of the (Component Description) class.
2. Fill in the class with appropriate information identifying the component that is to be registered.
3. Call the component register function of the Product Registry class.

The register function takes the Component Description as an argument. This function does all the required steps to open the persistent storage, acquire a lock, add the entry to the persistent storage, and free the lock.

Registering a component must address two possibilities. The registration:

- Is a new installation. In this case, the component is issued a new unique identifier.
- Is performed during an overinstall. In this case, the new component matches one already installed and must overlay the component in the PR. Additionally, the previously registered component's relationships must be updated with the new overlaid component information.

Further, the registration of each component must provide support for component dependencies. Components can be dependent on other components and may themselves be a dependency of other components. Both of these relationships must be able to be expressed in the PR.

The PR query capability must be able to determine whether:

- A component is already installed on the system and, if it is, whether the version of the registered component is the same as the one being installed.
- Any components depend on the one being installed and, if so, what the dependencies are.
- This component depends on the presence of other components and, if so, which ones.

As part of component registry, circular dependencies must be guarded against.

4.1.5.1 Multiple Versions of a Product Component

As components are maintained, it is sometimes necessary to break backward compatibility. In this situation, clients of the older component are not compatible with the new component. Also, clients dependent on the new component will not work with the older component.

If a user installs two products, each requiring a different version of the component, multiple versions of the required component must be installed in order to allow both products to work simultaneously.

This illustrates the need for the Product Registry to support registration of multiple versions of any component.

4.1.5.2 Multiple Instances of a Component

In some cases, it is desirable to have more than one copy of the same component. This may allow the system administrator to configure the components differently. The Product Registry should support multiple installed instances of components.

4.1.5.3 Product Registry Clients

A client of the Product Registry API is an application or service that imports the registry interface and uses that interface to interact with the registry.

4.1.6 Component Queries

Querying for components is the act of looking in the persistent storage for a component matching a given description. The get function insulates the caller from the details of opening and reading the persistent storage.

The get function takes as an argument a class with information describing the desired component. If the component is registered, a Component Description class is filled in and passed back to the

caller. If the specified component is not registered, a NULL value is returned.

A component can be identified for a query in several ways:

- **Component ID.** If a component identified by the specified ID is registered, the latest instance of the latest version of that component is returned.
- **Component ID, version, instance.** If a component identified by the specified ID and of the specified version is registered, the specified instance of that version of the component is returned.
- **Component ID, install location.** If a component identified by the specified ID is installed in the specified install location, that component is returned.
- **Unique name.** If a component identified by the specified name is registered, the latest instance of the latest version of that component is returned.
- **Unique name, version, instance.** If a component identified by the specified name and the specified version is registered, the specified instance of that version of the component is returned.
- **Unique name, install location.** If a component identified by the specified name is installed in the specified install location, that component is returned.

(See “Example: Public Class RegistryQuery” on page 75 for specifications.)

4.1.7 Component Removal

When a product is uninstalled, the associated components must be removed from the Product Registry. This section discusses only the API available for removing a component from the registry. A component is removed from the registry through a call to the Product Registry’s unregister method. This method takes a component description object as a parameter that is filled in to describe the component that is to be removed. The component description object must exactly match a registered component for it to be removed. The section “Product Registry Behavior” on

page 59 addresses when it is legal to remove a component from the registry.

4.1.8 Component Updates

The ability to modify a registered component is important because component dependencies are cross-referenced in the registry. The cross reference involves updating the required component's description to include the component that requires it (the dependent component). (See “Updating a Component” on page 60.)

It would be unfortunate to force a client to unregister a component and then register that component again in order to achieve a component modification. The behavior section will illustrate that there are situations in which a component cannot be removed or unregistered.

If a registered product component requires modification, it is done through a call to the register function.

The Component Description class is first obtained through a call to the get function. This ensures that the Component Description exactly matches the description of a currently registered component. After obtaining an exact component structure, the contents of the Component Description can be modified to achieve the desired update.

Some fields in the Component Description must not be modified, or the register call will result in a new Component Description being registered instead of an existing Component Description being updated.

The fields that must not be modified to achieve an update are:

- Component ID
- Install Location

An update can be guaranteed not to register a new component by first obtaining the Component Description class through the get function and then modifying the desired fields of that structure before calling the register function.

4.2 Product Registry Behavior

This section describes how the Product Registry behaves and how a client will use the Product Registry during installation, updates, and removal of a product.

4.2.1 Component Installation

The simplest case of Product Registry behavior is installing a component that has not previously been installed. In this case, the component is registered and then the component bits are installed on disk. This behavior occurs when installing a single instance of an installable component.

Installing a component that is dependent on another component is a more complex case. Before the component is installed the registry must be queried for the required component. If the required component does not exist in the registry, the installation will be terminated. However, if the required component does exist, the component is updated to reflect a new dependency: the dependency that the newly installed component has on the required component.

Care must be taken to ensure that components that indicate they support a particular version really fully support that version. Full support can be quite subtle since configuration files, header files, data files, library interfaces, interprocess communication and network communication protocols can all vary between versions. If not fully backwards compatible, a version must not be listed as a compatible version.

The Product Registry should handle the installation of multiple instances of the same component in different locations. This capability allows the same application to be configured differently. In this case, the instance number will differentiate between the otherwise identical components.

4.2.2 Updating a Component

A component is updated by registering the component. That is:

- A query is executed to locate the component.
- The component is modified and reregistered.

Note that the component ID and the install location cannot be modified. If either is modified, a new instance must be created in the registry, which results in the original component not being updated at all.

Updating a component must also address dependency issues. If a target component to be upgraded is required by other components, the update may break these dependent components if backward compatibility is an issue. A check must be made to determine if the upgrade is backward compatible with the versions that all of the dependent components expect. If the upgrade of is not backward compatible, the upgrade cannot proceed. However, if the upgrade is compatible, the upgraded component information must be merged with all the registered components. The upgraded registry entry must merge all of the application specific data, the dependent components, and all required components.

4.2.3 Removing a Component

The simplest case of Product Registry behavior is uninstalling a component that has no components dependent on it. Component dependency is determined by querying the Product Registry.

- If the query results indicate that the target component has no dependent components, the target component can be removed by unregistering it first and then removing its bits from the disk.
- If the query results indicate the target component has dependent children, those children (and their children) components must be recursively unregistered and removed first.
- If the query results indicate that another component is dependent on the component targeted for removal, target component removal must not be performed. In some

circumstances the registry may show that there are dependencies which in fact do not exist. For example, the files corresponding to the component may have been erased. In these cases, if it can be determined that the registry is inaccurate, there must be a way to remove components registered in error.

- If the query results indicate that the target component has required components, the association between the component targeted for removal and the required components must be removed. This Product Registry maintenance step must be performed so that components no longer appear as dependencies, which would erroneously prevent their removal from the system.

4.3 Product Registry Interfaces and Classes

4.3.1 Public Interface RegistryKey

4.3.1.1 Syntax

```
public interface RegistryKey
```

4.3.1.2 Description

Defines the interface for uniquely identifying a registerable component instance.

Member Summary	
Methods	
`getID()`	Returns the universal unique identifier for the component.
`getInstance()`	Returns the instance number assigned to the component.
`getVersion()`	Returns the version of the component.

Member Summary	
`setID(String)`	Sets the universal unique identifier for the component.
`setInstance(int)`	Sets the instance number of the component.
`setVersion(String)`	Sets the version of the component.

4.3.1.3 Methods

`getID()`

```
public java.lang.String getID()
```

Returns the universal unique identifier for the component. If the ID is not set on the component, may return null.

Returns: : The universal unique identifier string.

`getInstance()`

```
public int getInstance()
```

Returns the instance number assigned to the component. The instance number is assigned by the native registry. If the instance number is not set on the component, may return null.

Returns: : The instance number of the component.

`getVersion()`

```
public java.lang.String getVersion()
```

Returns the version of the component. If the version is not set on the component, may return null.

Returns: : The version string.

`setID(String)`

```
public void setID(java.lang.String id)
```

Sets the universal unique identifier for the component.

Parameters:
`id` - The universal unique identifier.

setInstance(int)

```
public void setInstance(int instance)
```

Sets the instance number of the component. The instance number is assigned by the native registry.

Parameters:
`instance` - The instance number of the component.

setVersion(String)

```
public void setVersion(java.lang.String version)
```

Sets the version of the component.

Parameters:
`version` - The string identifying the version of the component.

4.3.2 Public Interface RegistryComponent

4.3.2.1 Syntax

```
public interface RegistryComponent extends RegistryKey
```

4.3.2.2 Description

Defines the interface for manipulating a component in the product registry

.

Member Summary	
Methods	
`addChild(RegistryKey)`	Adds the specified child component to this component.
`addCompatibleVersion (String)`	Adds the specified version to the list of compatible versions.
`addDependentComponent (RegistryKey)`	Adds the specified component as a dependent component.
`addRequiredComponent (RegistryKey)`	Adds the specified component as a required component.
`getChildren()`	Returns this component's children RegistryKey objects.
`getCompatibleVersions()`	Returns an Enumeration of String objects describing which versions this component is backward compatible with.
`getComponentType()`	Returns the component type associated with this component.
`getData(String,String)`	Returns the application-specific data value for the given key.
`getDataKeys()`	Gets the list of all application-specific data keys.
`getDependentComponents()`	Returns the RegistryKey objects for the components that depend upon this component.
`getDisplayLanguages()`	Returns the languages for which display names have been set.

Member Summary	
`getDisplayName()`	Returns this component's displayName.
`getDisplayName(String)`	Returns this component's displayName in the specified language.
`getLocation()`	Returns the install directory of this component.
`getParent()`	Returns the RegistryKey object that describes the parent of this component.
`getRequiredComponents()`	Returns the RegistryKeys for the components that require this component.
`getUninstaller()`	Returns the command used to uninstall this component.
`getUniqueName()`	Returns the component’s unique name.
`getVendor()`	Returns the vendor of this component.
`isChild(RegistryKey)`	Returns true if the passed in RegistryKey is for a child of this RegistryComponent.
`removeChild(RegistryKey)`	Removes the specified child from this component.
`removeCompatibleVersion (String)`	Removes the specified version from the list of compatible versions.
`removeDependentComponent (RegistryKey)`	Removes the specified component from the list of dependent components.

Member Summary	
`removeRequiredComponent (RegistryKey)`	Removes the specified component from the list of required components.
`setComponentType (ComponentType)`	Sets the component type associated with this component.
`setData(String)`	Sets an application-specific key/value pair.
`setDependentComponents (Vector)`	Sets the list of RegistryKey objects for components that require this component.
`setDisplayName(String)`	Sets the displayName of this component.
`setDisplayName(String, String)`	Sets the displayName of this component for the specified language.
`setLocation(String)`	Sets the install directory of this component.
`setParent(RegistryKey)`	Sets the RegistryKey object that describes the parent of this component.
`setRequiredComponents (Vector)`	Sets the list of RegistryKeys for components that this component requires.
`setUninstaller(String)`	Sets the command used to uninstall this component.
`setUniqueName(String)`	Sets this component's unique name.
`setVendor(String)`	Sets the vendor of this component.

4.3.2.3 Methods

addChild(RegistryKey)

```
public void addChild(RegistryKey child)
```

Adds the specified child component to this component.

Parameters:
`child` - The new child component.

addCompatibleVersion(String)

```
public boolean addCompatibleVersion(java.lang.String
          version)
```

Adds the specified version to the list of compatible versions. This list indicates which versions this component is backward compatible with.

Parameters:
`version` - The compatible version to be added.

Returns: True on success; false otherwise.

addDependentComponent(RegistryKey)

```
public boolean addDependentComponent(RegistryKey
          dependentComponent)
```

Adds the specified component as a dependent component.

Parameters:
`dependentComponent` - the dependent component.

Returns: True if the specified component is added; false otherwise.

addRequiredComponent(RegistryKey)

```
public boolean addRequiredComponent(RegistryKey
          requiredComponent)
```

Adds the specified component as a required component.

Parameters:
`requiredComponent` - the required component.

Returns: True if the required component is added.

getChildren()

```
public RegistryKey[] getChildren()
```

Returns this component's children RegistryKey objects. If no children are set, may return null.

Returns: The children RegistryKeys.

getCompatibleVersions()

```
public java.lang.String[] getCompatibleVersions()
```

Returns an Enumeration of String objects describing which versions this component is backward compatible with. If no compatible versions are set, may return null.

Returns: The backward compatible versions in an enumeration.

getComponentType()

```
public ComponentType getComponentType()
```

Returns the component type associated with this component.

Returns: The ComponentType of this RegistryComponent.

getData(String,String)

```
public java.lang.String[] getData(java.lang.String
          key)
```

Returns the application-specific data value for the given key.

Parameters:
`key` - The key for the data to return.

Returns: The application-specific data as a string.

getDataKeys()

```
public java.lang.String[] getDatakeys
```

Gets the list of all application-specific data keys.

Returns: An array of string keys.

getDependentComponents()

```
public java.util.Vector getDependentComponents()
```

Returns the RegistryKey objects for the components that depend upon this component. If no dependent components are set, may return null.

Returns: A vector of RegistryKey components that depend upon this one.

getDisplayLanguages()

```
public java.lang.String[] getDisplayLanguages()
```

Returns the languages for which display names have been set. If no display names are set, may return null.

Returns: The display languages.

getDisplayName()

```
public java.lang.String getDisplayName()
```

Returns this component's displayName. If no display name is set, may return null.

Returns: The displayName of this component.

getDisplayName(String)

```
public java.lang.String getDisplayName
          (java.lang.String language)
```

Returns this component's displayName in the specified language. If no display name has been set for the specified language, may return null.

Parameters:

`language` - The 2 character language string.

Returns: The display name.

getLocation()

```
public java.lang.String getLocation()
```

Returns the install directory of this component. If no install location is set, may return null.

Returns: The install directory.

getParent()

```
public RegistryKey getParent()
```

Returns the RegistryKey object that describes the parent of this component. If this component does not have a parent, may return null.

Returns: The RegistryKey of the parent.

getRequiredComponents()

```
public java.util.Vector getRequiredComponents()
```

Returns the RegistryKeys for the components that require this component. If no components require this component, may return null.

Returns: The RegistryKeys of the components that require this component.

getUninstaller()

```
public java.lang.String getUninstaller()
```

Returns the command used to uninstall this component.

Returns: The uninstall command.

getUniqueName()

```
public java.lang.String[] getUniqueName()
```

Returns the component's unique name.

Returns: The unique name of the component.

getVendor()

```
public java.lang.String getVendor()
```

Returns the vendor of this component.

Returns: The vendor.

isChild(RegistryKey)

```
public boolean isChild(RegistryKey child)
```

Returns true if the passed in RegistryKey is for a child of this RegistryComponent.

Parameters:
`child` - the RegistryKey for the child

Returns: True if the RegistryKey is for a child of this component.

removeChild(RegistryKey)

```
public void removeChild(RegistryKey child)
```

Removes the specified child from this component.

Parameters:
`child` - The child component to remove.

removeCompatibleVersion(String)

```
public boolean removeCompatibleVersion
          (java.lang.String version)
```

Removes the specified version from the list of compatible versions. This list indicates which version this component is backward compatible with.

Parameters:
`version` - The compatible version to be removed.

Returns: True on success; false otherwise.

`removeDependentComponent(RegistryKey)`

```
public boolean removeDependentComponent(RegistryKey
          dependentComponent)
```

Removes the specified component from the list of dependent components.

Parameters:
`dependentComponent` - the dependent component to remove

Returns: True if the specified component is removed; false otherwise.

`removeRequiredComponent(RegistryKey)`

```
public boolean removeRequiredComponent(RegistryKey
          requiredComponent)
```

Removes the specified component from the list of required components.

Parameters:
`requiredComponent` - the required component to remove

Returns: True if the specified component is removed; false otherwise.

`setComponentType(ComponentType)`

```
public void setComponentType(ComponentType
          componentType)
```

Sets the component type associated with this component.

Parameters:
`ComponentType` - The type of this component (product, feature, component).

setData(String)

```
public boolean setData(java.lang.String key,
          java.lang.String value)
```

Sets an application-specific key/value pair.

Parameters:

`key` - The key used to identify/access the data.

`value` - The data value as a string.

Returns: True if the data was set; otherwise, false.

setDependentComponents(Vector)

```
public void setDependentComponents(java.util.Vector
          dependentComponents)
```

Sets the list of RegistryKey objects for components that require this component.

Parameters:

`dependentComponents` - A vector of RegistryKeys for component that depend upon this component.

setDisplayName(String)

```
public void setDisplayName(java.lang.String
          displayName)
```

Sets the displayName of this component.

Parameters:

`displayName` - The displayName of this component.

setDisplayName(String, String)

```
public void setDisplayName(java.lang.String
          language, java.lang.String displayName)
```

Sets the displayName of this component for the specified language.

Parameters:

`language` - The language to which the specified display name applies.

`displayName` - The displayName of this component.

setLocation(String)

```
public void setLocation(java.lang.String location)
```

Sets the install directory of this component.

Parameters:

`location` - The directory this component is installed in.

setParent(RegistryKey)

```
public void setParent(RegistryKey parent)
```

Sets the RegistryKey object that describes the parent of this component.

Parameters:

`parent` - the RegistryKey of the object that it is to be set as this component's parent

setRequiredComponents(Vector)

```
public void setRequiredComponents(java.util.Vector
          requiredComponents)
```

Sets the list of RegistryKeys for components that this component requires.

Parameters:

`requiredComponents` - The list of RegistryKeys for components depending on this component.

setUninstaller(String)

```
public void setUninstaller(java.lang.String
          uninstaller)
```

Sets the command used to uninstall this component.

Parameters:
`uninstaller` - The uninstall command.

`setUniqueName(String)`

```
public void setUniqueName(java.lang.String
           uniqueName)
```

Sets this component's unique name.

Parameters:
`uniqueName` - The unique name of the component.

`setVendor(String)`

```
public void setVendor(java.lang.String vendor)
```

Sets the vendor of this component. If the vendor of this component is not set, may return null.

Parameters:
`vendor` - The vendor of this component.

4.3.3 Example: Public Class RegistryQuery

4.3.3.1 Syntax

```
public class RegistryQuery extends java.lang.Object
   implements java.io.Serializable
```

```
java.lang.Object
   |
  +--productregistry.RegistryQuery
```

4.3.3.2 Description

This class provides a holder for information required to process a query of the Product Registry

.

Member Summary	
Constructors	
`RegistryQuery()`	No arg constructor.
Methods	
`getID()`	Returns the universal unique identifier this RegistryQuery object will be used to filter for.
`getInstance()`	Returns the component instance number this RegistryQuery will filter for.
`getLocation()`	Returns the install directory of the component this RegistryQuery will filter for.
`getUniqueName()`	Returns the unique name this RegistryQuery will be used to filter for.
`getVersion()`	Returns the component version this RegistryQuery will filter for.
`setID(String)`	Sets the universal unique identifier this RegistryQuery will be used to filter for.
`setInstance(int)`	Sets the component instance number this RegistryQuery will filter for.
`setLocation(String)`	Sets the install directory of the component this RegistryQuery will filter for.
`setUniqueName(String)`	Sets the unique name this RegistryQuery will be used to filter for.
`setVersion(String)`	Sets the component version this RegistryQuery will filter for.

4.3.3.3 Constructors

`RegistryQuery()`

```
public  RegistryQuery()
```

No arg constructor.

4.3.3.4 Methods

`getID()`

```
public final java.lang.String getID()
```

Returns the universal unique identifier this RegistryQuery object will be used to filter for.

`getInstance()`

```
public final int getInstance()
```

Returns the component instance number this RegistryQuery will filter for.

`getLocation()`

```
public java.lang.String getLocation()
```

Returns the install directory of the component this RegistryQuery will filter for.

Returns: The install directory.

`getUniqueName()`

```
public java.lang.String getUniqueName()
```

Returns the unique name this RegistryQuery will be used to filter for.

Returns: The unique name of the desired component.

`getVersion()`

```
public final java.lang.String getVersion()
```

Returns the component version this RegistryQuery will filter for.

setID(String)

```
public final void setID(java.lang.String id)
```

Sets the universal unique identifier this RegistryQuery will be used to filter for.

Parameters:
`id` - The universal unique identifier.

setInstance(int)

```
public final void setInstance(int instance)
```

Sets the component instance number this RegistryQuery will filter for.

Parameters:
`instance` - The instance number of the component being queried.

setLocation(String)

```
public void setLocation(java.lang.String location)
```

Sets the install directory of the component this RegistryQuery will filter for.

Parameters:
`location` - The directory the desired component is installed in.

setUniqueName(String)

```
public void setUniqueName(java.lang.String
          uniqueName)
```

Sets the unique name this RegistryQuery will be used to filter for.

Parameters:
`uniqueName` - The abbreviated form of the component name.

setVersion(String)

```
public final void setVersion(java.lang.String
          version)
```

Sets the component version this RegistryQuery will filter for.

Parameters:

`version` - The string identifying the version of the desired component.

This page left blank intentionally.

Chapter 5 API Specification: Actions and Rules

This section describes the Actions, Rules, and Describable interface APIs.

The Action interface defines the API required for an Action. The Rule associated with an Action determines whether to execute the action or not. If the rule evaluates to true, the action is executed. Action and Rules are packages that contain interfaces and classes. Their sections in this chapter describe the package interface, class, or exception of the API in terms of functionality and use.

The Describable interface provides a means to annotate the classes derived from the Action and Rules interfaces. Both Action and Rules extend this interface.

Not all APIs are described; only those required in the specification. For the detailed, definitive description of the API, including complete field and method explanations, refer to the Javadoc generated from the source distribution.

5.1 Actions

This package contains the interfaces and classes required to implement actions which provide the capability to execute procedures. Actions provide a simple class that facilitates the execution of a diverse set of activities from file I/O to execution of external programs. Actions:

- Provide a standardized API that allows programmers to easily augment the Builder and resulting Installer with custom actions.
- Actions should be available within the context of a registerable install components such as the Action Unit (see “Public Interface Unit” on page 30 and “Example: Public Class ActionUnit” on page 34 for more information on Action Units.

This package defines the following two interface classes:

- *Actions*, which are created and "executed" in order to get work done. Each action is controlled by a *rule* which determines if the action is executed or not. In this way, logical branches can be assembled within a group of actions. For more information on rules see the description of the rule package, below.
- *ActionContainers*, which are used as containers in which Actions are assembled. The ActionGroups interface provides the ability to aggregate Actions.

The two listed classes are examples of action implementations. The ExecAction provides a means to call external programs from within the Runtime Installer. Used in conjunction with the ActionUnit (see “Example: Public Class ActionUnit” on page 34), an ExecAction can be used to invoke custom scripts and native installer programs as well as perform other external program calls from within the Runtime Installer. The AddWinRegKeyAction example provides a means of adding a Windows registry key.

The following table summarizes the package interfaces and classes.

Note: Support for Progress is voluntary. The method supportsProgress is present to provide this flexibility.

Class Summary	
Interfaces	
`Action`	Provides an interface which defines what an `Action` can and should do.
`ActionContainer`	Provides an interface for grouping `Actions` together to create a new `Action`.
Classes	

Class Summary	
`AddWinRegKeyAction`	An action that can be used to make Windows registry keys.
`ExecAction`	Executes an external program in a subprocess.

5.1.1 Public Interface Action

5.1.1.1 Syntax

```
public interface Action extends Describable
```

5.1.1.2 Description

Provides an interface which defines what an `Action` can and should do. `Action`s are meant to be used similarly to functions; when executed, an `Action` will perform certain operations. `Action`s can consist of zero to many `Action`s or functions.

Member Summary	
Methods	
`cancel()`	Cancels this Action.
`evaluateRule()`	Evaluates the Rule if it has been set.
`execute(ProgressContext)`	Executes this action.
`getRule()`	Gets the Rule that governs this Action.
`initializeProgress (ProgressContext)`	Initializes the specified progress context for the purpose of reporting progress during the execution of this Action.

Member Summary	
`resume()`	Resumes this Action after suspend() has been called.
`setRule(Rule)`	Sets the Rule that governs this Action.
`supportsProgress()`	Indicates whether the action supports progress or not.
`suspend()`	Suspends this Action until resume() is called.

5.1.1.3 Methods

cancel()

```
public void cancel()
```

Cancels this Action. This implies this Action is currently running in another thread. If this Action is called again, it runs from the beginning, rather than from where it left off.

evaluateRule()

```
public boolean evaluateRule()
```

Evaluates the Rule if it has been set. If the rule is not set, returns true.

Returns: Boolean true if rule evaluates as true; otherwise false.

execute(ProgressContext)

```
public ProgressContext execute (ProgressContext
          progress)
```

Executes this action. Execution will depend on the outcome of evaluateRule(). If evaluteRule() returns true, the method is not executed, if it returns false, the method is executed.

Parameters:

`progress` - the initialized ProgressContext

Exception:

An exception is thrown for any errors that are encountered.

Returns:

If null is not passed in, an updated ProgressContext; else, null.

getRule()

```
public Rule getRule()
```

Gets the rule that governs this action. If the rule is not set, may return null.

Returns: The rule previously set.

initializeProgress(ProgressContext)

```
public ProgressContext initializeProgress
          (ProgressContext progress)
```

Initializes the specified progress context for the purpose of reporting progress during the execution of this action.

Parameters:

`progress` - the ProgressContext to initialize

resume()

```
public void resume()
```

Resumes this action after suspend() has been called.

setRule(Rule)

```
public void setRule(Rule rule)
```

Sets the rule that governs this action. Null is a valid value.

Parameters:

`rule` - the rule to set

supportsProgress()

```
public boolean supportsProgress()
```

Indicates whether the action supports progress or not. Actions which do not support progress still take a Progress object in the execute() method, but do not open or close any progress sections, so the progress object is unused.

Returns: True if the action reports progress; false if not.

suspend()

```
public void suspend()
```

Suspends this Action until resume() is called.

5.1.2 Public Interface ActionContainer

5.1.2.1 Syntax

```
public interface ActionContainer extends Action
```

5.1.2.2 Description

Provides an interface for grouping `Actions` together to create a new `Action`. The resulting `Actions` can be used as any other `Action`.

Member Summary	
Methods	
`addChild(Action)`	Adds a new Action to this ActionContainer.
`contains(Action)`	Determines whether an Action is in this ActionContainer.
`getChildren()`	Gets the Actions in this ActionContainer.

Member Summary	
`removeChild(Action)`	Removes a previously added Action from this ActionContainer.
`removeChildren()`	Removes all of the Actions from this ActionContainer.
`setChildren(Action[])`	Sets the entire list of child actions to the given list.

5.1.2.3 Methods

addChild(Action)

```
public void addChild(Action action)
```

Adds a new Action to this ActionContainer. The order in which Actions are added determines when they are started. First Action in is first to run, second Action is second and so on.

Parameters:
`action` - the action to set

contains(Action)

```
public boolean contains(Action action)
```

Determines whether an Action is in this ActionContainer.

Returns: Boolean true if this ActionContainer contains the given Action.

getChildren()

```
public Action[] getChildren()
```

Gets the Actions in this ActionContainer. If no actions are added to this container, may return null.

Returns: Action[] the Action array

removeChild(Action)

```
public void removeChild(Action action)
```

Removes a previously added Action from this ActionContainer. This does not affect the order in which the remaining Actions are started.

Parameters:
`action` - the action to remove

Throws:
`javax.jifi.action.ActionNotFoundException` - if the specified Action is not in the container.

removeChildren()

```
public void removeChildren()
```

Removes all of the Actions from this ActionContainer.

setChildren(Action[])

```
public void setChildren(Action[] actionSet)
```

Sets the entire list of child actions to the given list.

Parameters:
`actionSet` - the array of actions to use as child actions

5.1.3 Example: Public Class ExecAction

5.1.3.1 Syntax

```
public class ExecAction extends SimpleAction

java.lang.Object
   |
  +--action.SimpleAction
        |
        +--action.ExecAction
```

5.1.3.2 Description

Executes an external program in a subprocess. getProcess() is used to get the ProcessProxy object of the subprocess. This class uses the ExecCardService to generate a process object.

Member Summary	
Constructors	
`ExecAction()`	No arg constructor.
`ExecAction(String)`	Constructs the ExecAction based upon the supplied command.
Methods	
`cleanupProperties()`	Cleans up the properties used to hold the user variables.
`execute (ProgressContext)`	Executes the associated command.
`getCommand()`	Returns the command string for this action.
`getCommandParam()`	Returns the parameter to the command string.
`getErrorStream()`	Returns the InputStream object for the associated stderr of the subprocess.
`getExecutable()`	Returns the executable filename needed to run the exec action.
`getExitStatus()`	Returns the exit status of the resulting subprocess.
`getInputStream()`	Returns the InputStream object for the associated stdout of the subprocess.

Member Summary	
`getOutputStream()`	Returns the OutputStream object for the associated stdin of the subprocess.
`getProcess()`	The Runtime object's exec() call produces a Process object.
`getProvideExecutable()`	Returns true if the provideExecutable check box is checked.
`getProvider()`	Gets the Provider for this action.
`getSource()`	Returns the source directory of the executable.
`getUserVariableName()`	Returns the variable name that holds the return status.
`initializeProgress (ProgressContext)`	Initializes the ProgressContext class for this execution.
`setCommand(String)`	Sets the command string for this action.
`setCommandParam (String)`	Sets parameter to the command string.
`setExecutable(String)`	Sets the executable filename needed to run the exec action.
`setProvideExecutable (String)`	Sets the boolean value of provideExecutable.
`setProvider (ActionFileProvider)`	Sets the Provider for this action.
`setSource(String)`	Sets the source directory of the executable.
`setTotalTime(long)`	Sets the estimated total time it will take to complete this command.

Member Summary	
`setUserVariableName (String)`	Sets the variable name that holds the return status.
`verify()`	Verifies the integrity of this object.

5.1.3.3 Constructors

ExecAction()

```
public  ExecAction()
```

No arg constructor.

ExecAction(String)

```
public  ExecAction(java.lang.String command)
```

Construct the ExecAction based upon the supplied command. The command must contain the executable and any arguments passed to the executable.

Parameters:
`command` - the command and arguments to be executed

5.1.3.4 Methods

cleanupProperties()

```
public void cleanupProperties()
```

Cleans up the properties used to hold the user variables.

Specified By: `PropertySource.cleanupProperties() in interface property.PropertySource`

execute(ProgressContext)

```
public ProgressContext execute(ProgressContext
          progress)
```

Executes the associated command.

Specified By: `Action.execute(ProgressContext)` in `interface Action`

Overrides: `SimpleAction.execute(ProgressContext)` in `class SimpleAction`

Parameters:
`progress` - the ProgressContext object for this command

Returns: The updated ProgressContext object

getCommand()

```
public java.lang.String getCommand()
```

Returns the command string for this action.

Returns: String the command to be executed

getCommandParam()

```
public java.lang.String getCommandParam()
```

Returns the parameter to the command string.

Returns: String the string containing the parameter to the command string

getErrorStream()

```
public java.io.InputStream getErrorStream()
```

getExecutable()

```
public java.lang.String getExecutable()
```

Returns the executable filename needed to run the exec action.

Returns: String the string containing the executable filename

getExitStatus()

```
public int getExitStatus()
```

Returns the exit status of the resulting subprocess.

Returns: The integer exit status of the subprocess.

getInputStream()

```
public java.io.InputStream getInputStream()
```

Returns the InputStream object for the associated stdout of the subprocess.

Returns: The InputStream object associated with the subprocess.

getOutputStream()

```
public java.io.OutputStream getOutputStream()
```

Returns the OutputStream object for the associated stdin of the subprocess.

Returns: The OutputStream object associated with the subprocess.

getProcess()

```
public ProcessProxy getProcess()
```

The Runtime object's exec() call produces a Process object. This object describes the process created to execute the command. This method returns that object.

Returns: The object created by this action's execute() method

getProvideExecutable()

```
public java.lang.Boolean getProvideExecutable()
```

Returns true if the provideExecutable check box is checked.

Returns: Boolean true if the provideExecutable box is checked.

getProvider()

```
public ActionFileProvider getProvider()
```

Gets the Provider for this action.

Returns: The ActionFileProvider for this action.

getSource()

```
public java.lang.String getSource()
```

Returns the source directory of the executable.

Returns: The source directory that contains the executable.

getUserVariableName()

```
public java.lang.String getUserVariableName()
```

Returns the variable name that holds the return status.

Returns: String variable name that the user set in the customizer.

initializeProgress(ProgressContext)

```
public ProgressContext initializeProgress
          (ProgressContext progress)
```

Specified By:
`Action.initializeProgress(ProgressContext) in interface Action`

Overrides: `SimpleAction.initializeProgress (ProgressContext) in class SimpleAction`

Parameters:
`progress` - the ProgressContext object to initialize

Returns: The initialized ProgressContext object.

setCommand(String)

```
public void setCommand(java.lang.String command)
```

Sets the command string for this action.

Parameters:

`command` - the string containing the programs and parameters to execute

setCommandParam(String)

```
public void setCommandParam(java.lang.String
          commandParam)
```

Sets parameter to the command string.

Parameters:

`commandString` - the string containing the parameter to the command string

setExecutable(String)

```
public void setExecutable(java.lang.String
          executable)
```

Sets the executable filename needed to run the exec action.

Parameters:

`executable` - the executable filename

setProvideExecutable(Boolean)

```
public void setProvideExecutable(java.lang.Boolean
          provideExecutable)
```

Sets the boolean value of provideExecutable.

Parameters:

`provideExecutable` - the boolean value of provideExecutable

setProvider(ActionFileProvider)

```
public void setProvider(ActionFileProvider provider)
```

Sets the Provider for this action.

Parameters:

`provider` - the provider to set

setSource(String)

```
public void setSource(java.lang.String source)
```

Sets the source directory of the executable.

Parameters:

`source` - the source directory in which the executable resides.

setTotalTime(long)

```
public void setTotalTime(long milliseconds)
```

Sets the estimated total time it will take to complete this command. There is no way reporting progress steps from the command itself and since the command is run in a separate process this action interrogates the exit status and continues to report status until it completes. It will check every 1/4 second.

Parameters:

`milliseconds` - the number of milliseconds it is estimated this command will take to complete

setUserVariableName(String)

```
public void setUserVariableName(java.lang.String
          variableName)
```

Sets the variable name that holds the return status.

Parameters:

`variableName` - that's set in the customizer

verify()

```
public verification.UserMessage[] verify()
```

Specified By: verification.Verifiable.verify() in interface verification.Verifiable

Overrides: `SimpleAction.verify()` in class `SimpleAction`

Returns: The array of user messages if the verification fails.

5.1.4 Example: Public Class AddWinRegKeyAction

5.1.4.1 Syntax

```
public class AddWinRegKeyAction extends WinRegAction

java.lang.Object
  |
  +--action.SimpleAction
        |
        +--action.WinRegAction
              |
              +--action.AddWinRegKeyAction
```

5.1.4.2 Description

An action that can be used to make Windows registry keys. Optionally, it can also add a String value to the new key. It is assumed to be acceptable if the key and/or value already exists.

Member Summary	
Constructors	
`AddWinRegKeyAction()`	Constructs an empty AddWinRegAction.
`AddWinRegKeyAction (int, String, boolean)`	Constructs this Action with the given rootKey (HKEY) constant and subkey name.
`AddWinRegKeyAction (int, String, boolean, String, String)`	Constructs this Action with the given rootKey (HKEY) constant, subkey name, valueName, and value.
Methods	
`execute (ProgressContext)`	Executes the register key action.
`getValue()`	Retrieves the value string.
`getValueName()`	Retrieves the value name.

Member Summary	
`getVolatile()`	Retrieves the volatile flag.
`initializeProgress (ProgressContext)`	Initializes the ProgressContext class for this execution.
`setValue(String)`	Sets the String to be associated with the optional new value when the key is added to the registry.
`setValueName(String)`	Sets the value name.
`setVolatile(boolean)`	Sets the volatile flag.
`verify()`	Verifies the integrity of this object.

5.1.4.3 Constructors

AddWinRegKeyAction()

```
public  AddWinRegKeyAction()
```

Constructs an empty AddWinRegAction.

AddWinRegKeyAction(int, String, boolean)

```
protected  AddWinRegKeyAction(int rootKey,
           java.lang.String subKeyName, boolean
           makeVolatile)
```

Constructs this Action with the given rootKey (HKEY) constant and subkey name.

Parameters:

`rootKey` - the root key (HKEY) constant

`subKeyName` - the remaining key name (subkey)

`volatileFlag` - true to make the new entry volatile in the registry

AddWinRegKeyAction(int, String, boolean, String, String)

```
protected  AddWinRegKeyAction(int rootKey,
           java.lang.String subKeyName, boolean
           makeVolatile, java.lang.String valueName,
           java.lang.String value)
```

Constructs this Action with the given rootKey (HKEY) constant, sub-key name, valueName, and value. This constructor is used to create a "combination" action that will both create a key and add a String value to it at the same time.

Parameters:

`rootKey` - the root key (HKEY) constant

`subKeyName` - the remaining key name (subkey)

`makeVolatile` - true to make the new entry volatile in the registry

`valueName` - the optional name of a value to add

`value` - the String value to assign to the new value

5.1.4.4 Methods

execute(ProgressContext)

```
public ProgressContext execute(ProgressContext
           progress)
```

Overrides: `SimpleAction.execute(ProgressContext) in class SimpleAction`

Parameters:

`progress` - the ProgressContext object for this command

Returns: The updated ProgressContext object.

getValue()

```
public java.lang.String getValue()
```

Retrieves the value string.

Returns: Value string.

getValueName()

```
public java.lang.String getValueName()
```

Retrieves the value name.

Returns: value name

getVolatile()

```
public boolean getVolatile()
```

Retrieves the volatile flag.

Returns: True if registry key is to be volatile; false otherwise.

initializeProgress(ProgressContext)

```
public ProgressContext initializeProgress
          (ProgressContext progress)
```

Initializes the ProgressContext class for this execution.

Overrides: `SimpleAction.initializeProgress (ProgressContext) in class SimpleAction`

Parameters:
`progress` - the ProgressContext object to initialize

Returns: The initialized ProgressContext object.

setValue(String)

```
public void setValue(java.lang.String value)
```

Sets the String to be associated with the optional new value when the key is added to the registry.

Parameters:
`value` - value string

setValueName(String)

```
public void setValueName(java.lang.String name)
```

Sets the value name.

Parameters:
`name` - value name

setVolatile(boolean)

```
public void setVolatile(boolean flag)
```

Sets the volatile flag.

Parameters:
`flag` - true if registry key is to be volatile; false otherwise

verify()

```
public verification.UserMessage[] verify()
```

Overrides: `WinRegAction.verify() in class WinRegAction`

Returns: `The array of user messages if the verification fails.`

5.2 Rules

Rules provide containers used to implement logical evaluators that are used with Actions. Rules provide a simple class that facilitates the gating of Actions based upon a simple evaluation and boolean return. Their implementation should provide a standardized means to evaluate logical expressions and incorporate the objects into other classes as a means of providing logic control.

This package defines two interfaces:

- Rule, which provides a single operation, evaluate(). Rules are contained in Action objects. Actions evaluate their Rule object before the body of their execute() method is attempted.
- RuleSet, which provides the ability to aggregate rules into more complex arrangements. For instance, multiple rules ANDed together comprise the AndRule object.
- In addition to the public interfaces Rule and RuleSet, this section includes WinRegKeyExists as a Rule class example.

Rules are contained in Action objects. Actions evaluate their Rule object before the body of their execute() method is attempted.

The following table summarizes the package classes.

Class Summary	
Interfaces	
`Rule`	Provides a mechanism for introducing logic into other components such as Actions.
`RuleSet`	Defines a collection of Rules.
Classes	
`WinRegKeyExists`	Determines whether a Windows Registry key exists.

5.2.1 Public Interface Rule

5.2.1.1 Syntax

```
public interface Rule extends Describable
```

5.2.1.2 Description

Provides a mechanism for introducing logic into other components such as Actions.

Member Summary	
Methods	
`evaluate()`	Evaluates this Rule.
`evaluate(Object)`	Evaluate this Rule based upon the supplied target object.

5.2.1.3 Methods

evaluate()

```
public boolean evaluate()
```

Evaluates this Rule.

Returns: Boolean true if rule evaluates true, otherwise false.

evaluate(Object)

```
public boolean evaluate(java.lang.Object target)
```

Evaluate this Rule based upon the supplied target object.

Parameters:

`target` - an object that will provide information toward the evaluation of this rule.

Returns: Boolean true if the rule evaluates to true, otherwise false.

5.2.2 Public Interface RuleSet

5.2.2.1 Syntax

```
public interface RuleSet
```

5.2.2.2 Description

Defines a collection of Rules. With this class, Rules can be combined to make new Rules, for example with an AndRule or OrRule.

Member Summary	
Methods	
`addRule(Rule)`	Adds a Rule to the RuleSet.
`getRules()`	Gets the Rules in the RuleSet.
`removeRule(Rule)`	Removes a Rule from the RuleSet.
`setRules(Rule[])`	Sets the Rules in the RuleSet.

5.2.2.3 Methods

addRule(Rule)

```
public void addRule(Rule rule)
```

Adds a Rule to the RuleSet.

Parameters:

`rule` - the rule to add

Throws:

`java.lang.IllegalArgumentException` - if the specified parameter is null.

getRules()

```
public Rule[] getRules()
```

Gets the Rules in the RuleSet. If no rules are set, may return null.

Returns: Rule[] the Rules in the RuleSet

removeRule(Rule)

```
public void removeRule(Rule rule)
```

Removes a Rule from the RuleSet.

Parameters:
`rule` - the rule to remove

Throws:
`java.lang.IllegalArgumentException` - if the specified parameter is null.

setRules(Rule[])

```
public void setRules(Rule[] ruleSet)
```

Sets the Rules in the RuleSet.

Parameters:
`ruleSet` - the array of Rules to be put in the RuleSet

Throws:
`java.lang.IllegalArgumentException` - if the specified parameter is null.

5.2.3 Example: Public Class WinRegKeyExists

5.2.3.1 Syntax

```
public class WinRegKeyExists extends WinRegRuleAdaptor

java.lang.Object
  |
  +--rule.RuleAdaptor
        |
        +--rule.WinRegRuleAdaptor
              |
              +--rule.WinRegKeyExists
```

5.2.3.2 Description

Determines whether a Windows Registry key exists.

Member Summary	
Constructors	
`WinRegKeyExists()`	Constructs this rule with no properties.
`WinRegKeyExists(int, String)`	Constructs this Rule with the given rootKey (HKEY) constant and subkey name.
Methods	
`evaluate()`	Evaluates if the registry key exists.

5.2.3.3 Constructors

WinRegKeyExists()

```
public  WinRegKeyExists()
```

Constructs this rule with no properties.

WinRegKeyExists(int, String)

```
public  WinRegKeyExists(int rootKey,
           java.lang.String subKeyName)
```

Constructs this Rule with the given rootKey (HKEY) constant and subkey name.

Parameters:

`rootKey` - the root key (HKEY) constant

`subKeyName` - the remaining key name (subkey)

5.2.3.4 Methods

evaluate()

```
public boolean evaluate()
```

Evaluates if the registry key exists.

Overrides: `RuleAdaptor.evaluate()` in class `RuleAdaptor`

Returns: Boolean true if rule evaluates true, otherwise false.

5.3 Describable

This interface provides a means to identify and annotate other classes in the API.

5.3.1 Public Interface Describable

5.3.1.1 Syntax

```
public interface Describable
```

5.3.1.2 Description

Provides the abstraction for any object that is addressable by the builder user to set and get descriptive text to/from the target object.

Member Summary	
Methods	
`setDescription (String description)`	Sets the specified description.
`getDescription()`	Gets the specified description.

5.3.1.3 Methods

setDescription(String description)

```
public void setDescription(String description)
```

Sets the specified description.

Parameters:

`String description` - descriptive text for the target object.

getDescription()

```
public String getDescription()
```

Gets the specified description.

Returns: The descriptive string.

Appendix A Glossary

Term	Description
Action	Any command or set of commands which performs some higher-level command.
ActionContainer	Any group of Actions which is to be executed together in some particular fashion as though it were one Action.
ActionGroup	A particular implementation of ActionContainer in which all the Actions therein contained are to be executed in order exactly once.
ActionUnit	A Reference Implementation class that provides a means for packaging an action into a registerable component and placing it in the Product Definition. The install() method calls the underlying Action's execute () method.
component	Broad term used to indicate any object in the product definition.

Term	Description
Feature	A feature is the smallest end-user selectable unit of software. Every product must have at least one feature. Features consist of one or more Product Components. Features organize files into logical units that can then be selected by the user. Typically, the installation keeps features under the convenience of the Install Set thereby simplifying the end user experience. When the user selects a 'Custom' install, features are presented and chosen by the user. Features can also contain any number of other features. Features can share components.
FileSetUnit	A Reference Implementation class that provides a collection of files that can be installed as a single unit. This unit can also be parented by a Product Component and thus registered as one unit.
Install Set	A product contains at least one install set. It is the minimum shippable unit of a software installation. An install set is a convenience aggregation for the end-user. Some examples of install sets include: Typical, Custom, Minimal, Maximum etc. Install sets are selectable at the end-user installation. Install sets comprise a set of features. Because they are only a convenience, install sets are not uniquely identified nor are they registered and tracked.

Term	Description
Installable Unit	The bottom of the Product Definition hierarchy. A generic term that describes any type of unit assigned to a Product Component. The Reference Implementation contains two unit implementations: the FileSetUnit and the ActionUnit.
Product	The top of a complete installation. A product contains at least one install set. It is the minimum shippable unit of a software installation. (Every Product Definition must contain at least one product.) Products consist of one or more features. Products have unique codes and names describing them. These product names represent the names presented to a user query about the products installed on the system. Product names and codes are stored in the registry. The features that they own are tracked with them.

Term	Description
Product Component	Product Components are the containers for files and actions. Product Components are created by installer builder developers. They are an organizational aggregation for the developer. A Product Component is the smallest installable piece that a developer can install. It is a collection of installable units. When the install or uninstall method is called the entire contents of the Product Component are installed or removed. Typically, only the install developer works at the component level. Product Components represent the smallest registerable class of the Product Definition and are contained in Features. Each Feature must have at least one Product Component.
Rule	A logical statement which can evaluate to true or false and which is used to determine whether some Action will be executed. Each Action, including ActionContainers, has a Rule which determines whether it should be executed.
RuleSet	A composite of rules which, combined in a certain way as defined by the RuleSet, evaluates to true or false, and can be used as if it were one Rule.
Suite	The top of a composite Product Definition. Used to aggregate products. Not all Product Definitions contain the Suite level.
Universal Unique Identifier (UUID)	A label used to uniquely identify each product component.