IBM

A First Look at Solution Installation for Autonomic Computing

- Enable autonomic solution installation
- Learn about the key technologies
- Get started today

Bart Jacob
Sudipto Basu
Amit Tuli
Patricia Witten

Redbooks

ibm.com/redbooks

International Technical Support Organization

A First Look at Solution Installation for Autonomic Computing

July 2004

SG24-7099-00

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (July 2004)

This edition applies to Version 1, Release 2, of the Solution Install for Autonomic Computing toolkt.

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

alphaWorks®	e-business on demand™	Redbooks (logo) ™
DB2®	ibm.com®	Redbooks™
developerWorks®	IBM®	WebSphere®

The following terms are trademarks of other companies:

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM Redbook provides a first look at the Solution Installation capability that is a key component of the IBM® Autonomic Computing initiative. Autonomic computing technologies enable complex computing systems to be self-configuring, self-healing, self-optimizing and self-protecting. One aspect of this is the ability to automate the change management processes associated with deploying and managing software required for business solutions.

The Solution Installation technologies that are described are evolving and will be enhanced over time. This publication provides the reader with an opportunity to understand the underlying concepts and facilities. This publication is based on Version 1.2 of the Solution Installation toolkit and describes both architectural concepts as well as specific examples that apply to the current version. We attempt to make clear when a key architectural concept is not fully implemented at this time.

Readers of this book will gain an understanding of the basic concepts and how the architecture described will provide business value as it becomes fully implemented and utilized by IBM and other software vendors.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Austin Center.

Bart Jacob is a Senior Consulting IT Specialist at IBM Corp - International Technical Support Organization, Austin Center. He has 24 years of experience providing technical support across a variety of IBM products and technologies, including communications, object-oriented software development, and systems management. He has over10 years of experience at the ITSO, where he has been writing IBM Redbooks™ and creating and teaching workshops around the world on a variety of topics. He holds a Masters degree in Numerical Analysis from Syracuse University.

Sudipto Basu has worked for IBM Global Services, India for more than four years. He has worked in large scale e-commerce Web site designing and building using J2EE specifications. He has extensive knowledge of J2EE, Web services, XML, WebSphere, LDAP and DB2®. He has worked with large financial clients and contributed in various online transactions and application

processes for them. He has a Masters degree in Computer Application from Banaras Hindu University, India.

Amit Tuli is working as a team member in the IBM India Research Lab. Amit has over four years of industry experience designing and developing business applications using a variety of IBM technologies and holds certifications in DB2, Java™ and Linux. He joined IBM in 2000. Currently, he is working on a data mining project. He has worked in the areas of J2EE applications and WebSphere® Business Components Development. He has a Masters in Computer Applications from GJ University in Haryana and a Bachelor of Computer Sciences from Delhi. Amit can be reached at tamit@in.ibm.com.

Patricia Witten has participated in software and hardware development projects as a consultant for IBM and other software development organizations. She has experience with IT Operations and Maintenance. Her recent work includes a consulting assignment for WebSphere Portal with the Department of Homeland Security at U.S. Customs.

Thanks to the following people that work on Solution Installation and related technologies in IBM for their contributions to this project:

Marcus Belvin
Colleen DeJong
Christine Draper
Toni Drapkin
Bryan Ellington
Randy George
Mike Gering
F Hackerman
Kathleen Harlow
Duy Huynh
Kesavan Kanaan
Doug Ledden
Joseph Lowengruber
Julia McCarthy
Caroline McDonnell
Denis McFadden
Chris Morgan
Bob Patten
Stan Smith

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. JN9B Building 003 Internal Zip 2834
  11400 Burnet Road
  Austin, Texas 78758-3493

1

Autonomic computing

Autonomic computing is the ability of an IT infrastructure to adapt to change in accordance with business policies and objectives. Quite simply, it is about freeing IT professionals to focus on higher-value tasks by making technology work smarter, with business rules guiding systems to be self-configuring, self-healing, self-optimizing, and self-protecting.

The focus of this publication is to introduce the reader to the Solution Installation aspect of the IBM Autonomic Computing initiative. Before we define Solution Installation and discuss it in detail, let us first describe autonomic computing in general.

1.1 What is autonomic computing

The term “autonomic” comes from an analogy to the autonomic central nervous system in the human body, which adjusts to many situations automatically without any external help. We walk up a flight of stairs and our heart rate increases. It is hot, we perspire. It's cold, we shiver. We do not tell ourselves to do these things, they just happen.

Similarly, the way to handle the problem of managing a complex IT infrastructure is to create computer systems and software that can respond to changes in the IT (and ultimately, the business) environment, so the systems can adapt, heal and protect themselves.

1.1.1 Guiding principles

The cost of technology continues to decrease, yet overall IT costs do not. With the expense challenges that many companies face, IT managers are looking for ways to improve the return on investment of IT by:

- Reducing total cost of ownership
- Improving quality of service
- Accelerating time to value
- Managing IT complexity

The IBM autonomic computing vision is for intelligent, open systems that:

- Manage complexity.
- “Know” themselves.
- Continuously tune themselves.
- Adapt to unpredictable conditions.
- Prevent and recover from failures.
- Provide a secure environment.

Autonomic computing systems consist of four attributes. As illustrated in the following 4-quadrant chart, they are:

- Self-configuring (able to adapt to changes in the system)
- Self-healing (able to recover from detected errors)
- Self-optimizing (able to improve use of resources)
- Self-protecting (able to anticipate and cure intrusions)

Figure 1-1 Autonomic computing attributes

Self-configuring

With the ability to dynamically configure itself, an IT environment can adapt immediately—with minimal intervention—to the deployment of new components or changes in the IT environment.

Self-healing

Self-healing IT environments can detect problematic operations (either pro-actively through predictions or otherwise) and then initiate corrective action without disrupting system applications. Corrective action could mean that a product alters its own state or influences changes in other elements of the environment. Day-to-day operations do not falter or fail because of events at the component level. The IT environment as a whole becomes more resilient as changes are made to reduce or help eliminate the business impact of failing components.

Self-optimizing

Self-optimization refers to the ability of the IT environment to efficiently maximize resource allocation and utilization to meet end users' needs with minimal intervention. In the near term, self-optimization primarily addresses the complexity of managing system performance. In the long term, self-optimizing components may learn from experience and automatically and pro-actively tune themselves in the context of an overall business objective.

Self-protecting

A self-protecting environment allows authorized people to access the right data at the right time and can take appropriate actions automatically to make itself less vulnerable to attacks on its runtime infrastructure and business data. A self-protecting IT environment can detect hostile or intrusive behavior as it occurs and take autonomous actions to make itself less vulnerable to unauthorized access and use, viruses, denial-of-service attacks and general failures.

1.2 Autonomic computing concepts

In an autonomic environment, components work together, communicating with each other and with high-level management tools. They can manage or control themselves and each other.

Components can manage themselves to some extent, but from an overall system standpoint, some decisions need to be made by higher level components that can make the appropriate trade-offs based on policies that are in place.

Let us start by looking at how a single entity is managed in an autonomic environment. The following figure represents the control loop that is the core of the autonomic architecture.

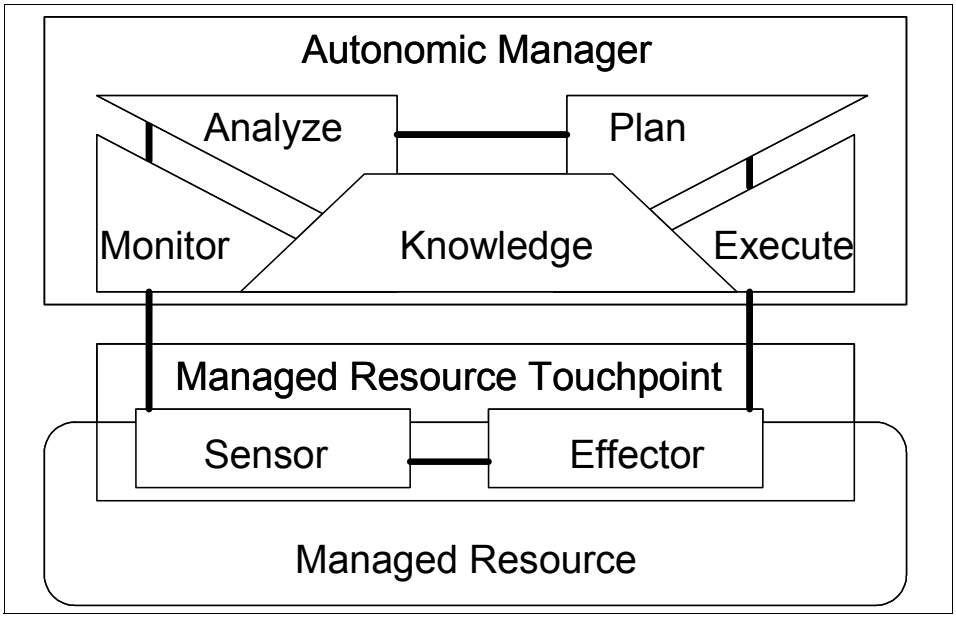

Figure 1-2 Autonomic computing control loop

1.2.1 Managed resources

The managed resource is a controlled system component. The managed resource can be a single resource or a collection of resources. The managed resource is controlled through its sensors and effectors.

The sensors provide mechanisms to collect information about the state and state transitions of an element. Sensors can be implemented using a set of *get* operations to retrieve information about the current state, or a set of management events (unsolicited, asynchronous messages or notifications) that flow when the state of the element changes in a significant way, or both.

The effectors are mechanisms that change the state (configuration) of an element. In other words, the effectors are a collection of *set* commands or application programming interfaces (APIs) that change the configuration of the managed resource in some way.

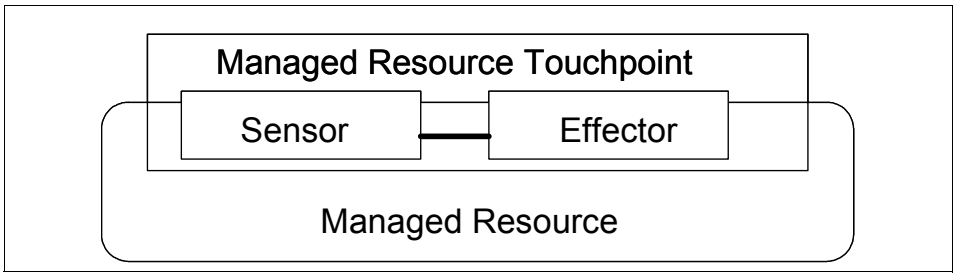

Figure 1-3 Sensors and effectors

The combination of sensors and effectors form the manageability interface (referred to as the touchpoint) that is available to an autonomic manager. The architecture encourages the idea that sensors and effectors are linked together. For example, a configuration change that occurs through effectors should be reflected as a configuration change notification through the sensor interface.

Web services can (and will) be used to implement sensor-effector functions. By utilizing a Web services architecture for communication to the managed resource touchpoint, current approaches to resource management can be reused and wrapped with a Web service.

Managed environments that can host software components are called hosting environments, and could be a typical operating system environment or a J2EE container, to name two.

1.2.2 Autonomic manager

The autonomic manager is a component that implements the control loop. The architecture dissects the loop into four parts that share knowledge:

- The monitor part provides the mechanisms that collect, aggregate, filter, manage, and report details (metrics and topologies) collected from a managed resource.
- The analyze part provides the mechanisms to correlate and model complex situations. These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations.
- The plan part provides the mechanisms to structure the action needed to achieve goals and objectives. The planning mechanism uses policy information to guide its work.
- The execute part provides the mechanisms that control the execution of a plan with considerations for on-the-fly updates.

The four parts work together to provide the control loop function. Figure 1-2 shows a structural arrangement of the parts—not a control flow. The bold line that connects the four parts should be thought of as a common messaging bus rather than a strict control flow. In other words, there can be situations where the plan part may ask the monitor part to collect more or less information. There could also be situations where the monitor part may trigger the plan part to create a new plan. The four parts collaborate using asynchronous communication techniques, like a messaging bus.

This architecture does not prescribe a particular management protocol or instrumentation technology. The architecture needs to work with the various computing technologies and standards that exist in the industry today, such as SNMP, Java Management Extensions (JMX), Distributed Management Task Force, Inc. (DMTF), Common Information Model (CIM), vendor-specific APIs or commands, as well as with new technologies that will emerge in the future.

Given the diversity of the approaches that already exist in the IT industry, this architecture endorses Web services techniques for sensors and effectors. These techniques encourage implementors to leverage existing approaches, and support multiple binding and marshalling techniques.

Multiple levels of autonomic managers

As implied by the references to complex scenarios above, environments will consist of many managed resources and many autonomic managers. For instance, in an ideal environment, each component of an application may be a managed resource with its own autonomic manager. However, the application as a whole could also be seen as a managed resource with its own autonomic

manager. In this case, the component-specific managers would have their own sensors and effectors that would allow them to report their status to and be controlled by the application's autonomic manager. In this way, the application could provide the intelligence or guidance to the individual components related to situations that affect the application as a whole. Policies would dictate what information and/or actions would be controlled solely within the component and what should be shared with or controlled by the application's autonomic manager.

Likewise, the application may be a part of a business solution consisting of many related applications. The capability for this architecture to scale through multiple levels of autonomic managers allows it to be implemented in a phased approach with benefits accruing at every level.

1.3 Value of autonomic computing

By enabling computer systems to have self-configuring, self-healing, self-optimizing and self-protecting features, autonomic computing is expected to have many benefits for business systems, such as reduced operating costs, lower failure rates, more security, and the ability to have systems that can respond more quickly to the needs of the business within the market in which they operate.

1.3.1 Software vendors

The primary business imperatives for enterprises today include driving down the total cost of ownership of the infrastructure while at the same time increasing user productivity.

Autonomic computing is core to enabling an automated environment, which is one of the three major characteristics of the e-business on demand™ Operating Environment; the other two being:

- Integrated, core business systems are linked within the business and across different enterprises.
- Virtualized, data and applications are managed centrally, via grid-like infrastructure that optimizes the use of computing capacity and delivers better performance.

The implications for an autonomic, on demand business approach are immediately evident: A network of organized, smart computing components can give clients what they need, when they need it, without a conscious mental or even physical effort.

ISVs will play a key role through building autonomic capabilities into their product sets, both hardware and software.

Once systems and networks begin to feature these attributes, IT professionals will be able to work at a higher level. Managers could set business goals, and computers would automatically set the IT actions needed to deliver them. For example, in a financial-trading environment, a manager might decide that trades have to be completed in less than a second to realize service and profitability goals. It would be up to software tools to configure the computer systems to meet those metrics.

As enterprises build infrastructures that support the dynamic needs of on demand computing, software products that conform to standards and are enabled for autonomic computing will be more attractive to the corporate decision makers choosing the components for their IT infrastructure.

1.3.2 Enterprises

Below are a few examples of the results delivered by implementing autonomic computing solutions with self-management characteristics:

- Operational efficiency

  As the IT infrastructure becomes more autonomic, executing on business policy becomes the focus of IT management. Management of the business and of IT will no longer be separate, and possibly conflicting, activities.

  Self-configuring and self-optimizing technologies drive efficiency in running and deploying new processes and capabilities.

- Supporting business needs with IT

  The actualization of self-configuring systems speeds the deployment of new applications required to support emerging business requirements.

  Self-healing capabilities help deliver the 24x7 availability required to keep businesses running.

- Workforce productivity

  Workforce productivity is enhanced when the focus is on management of business processes and policies, without the need to translate these needs into actions that separately manage supporting technology.

Systems that are self-managing free up IT resource to move from mundane system management tasks to focusing on working with users to solve business problems.

1.4 Core capabilities

In order for the autonomic managers and the managed resources in an autonomic system to work together, the developers of these components need a common set of capabilities. This section conceptually describes an initial set of core capabilities that are needed to build autonomic systems. These core capabilities include:

- Solution change management (the subject of this publication)
- Common systems administration
- Problem determination
- Autonomic monitoring
- Complex analysis
- Policy-based management
- Heterogeneous workload management

1.4.1 Solution Installation

Today there are a myriad of install, configure and maintenance mechanisms for software solutions. Having various mechanisms creates difficulties for IT departments installing software in complex systems environments due to the differences and idiosyncrasies of many system administration tools and distribution packaging formats. These problems are further compounded in a Web services environment, where application functionality can be composed dynamically. From an autonomic systems perspective, lack of solution knowledge inhibits important elements of self-configuring, self-healing, self-optimizing and self-protecting. Solution Installation not only provides change management capabilities but also builds and maintains solution knowledge.

A common solution knowledge capability eliminates the complexity introduced by many formats and many installation tools. By capturing installation and configuration information in a consistent manner, autonomic managers can share the facilities as well as information regarding the installed environment in contexts beyond installation, such as problem determination or optimization.

Solution Installation provides the underlying capabilities to:

- Define software installable units and their associated requisites in a standard way.
- Aggregate these installable units to create complex distributed computing solutions.
- Provide mechanisms to deploy these solutions on demand in an efficient and secure manner.
- Track the installed software and the complex relationships between the installable units.

Of course, to perform these deployments in a consistent manner across a variety of platforms requires open standards to be in place.

Managed environments that can host software components are called hosting environments, and could be a typical operating system environment or a J2EE container, to name two. The interface that will allow a conforming installation program or other systems management software to perform change management operations on these hosting environments is called a touchpoint and is implemented as a Web service.

The focus of this publication is Solution Installation. However, for completeness, we provide a summary of the other autonomic computing core capabilities in the following sections.

1.4.2 Common systems administration

Autonomic systems require common console technology to create a consistent human interface for the autonomic managers of the IT infrastructure. The common console capability provides a framework for reuse and consistent presentation for other autonomic core technologies.

The primary goal of a common console is to provide a single platform that can host all of the administrative console functions in server, software, and storage products in a manner that allows users to manage solutions rather than managing individual systems or products. Administrative console functions range from setup and configuration to solution runtime monitoring and control.

The value to the customer is reduced cost of ownership, attributable to more efficient administration, and reduced learning curves as new products and solutions are added to the autonomic system environment. By enabling increased consistency of presentation and behavior across administrative functions, the common console creates a familiar user interface that promotes reusing learned interaction skills versus learning new and different product-unique interfaces.

1.4.3 Problem determination

Autonomic managers take actions based on problems or situations they observe in the managed resource. Therefore, one of the most basic capabilities is being able to extract high quality data to determine whether or not a problem exists in managed resources. In this context, a problem is a situation in which an autonomic manager needs to take action. A major cause of poor quality information is the diversity in the format and content of the information provided by the managed resource.

To address this diversity of the data collected, a common problem determination architecture normalizes the data collected, in terms of format, content, organization and sufficiency. To do this, it defines a base of data that must be collected or created when a situation or event occurs. This definition includes information on both the kinds of data that must be collected as well as the format that must be used for each field collected. The problem determination architecture categorizes the collected data into a set of situations. Situations represent the state of an application such as starting or stopping. The technologies used to collect the autonomic data must be capable of accommodating legacy data sources (such as logs and traces) as well as data that is supplied using the standard format and categorization. To accommodate this legacy data, the architecture defines an adapter/agent infrastructure that will provide the ability to plug in an adapter to transform data from a component-specific format to the standard format, as well as sensors to control data collection.

1.4.4 Autonomic monitoring

Autonomic monitoring is a capability that provides an extensible runtime environment for an autonomic manager to gather and filter data obtained through sensors. Autonomic managers can utilize this capability as a mechanism for representing, filtering, aggregating and performing a range of analysis of sensor data. An autonomic manager using this autonomic monitoring functionality can help manage certain applications or resources more effectively through:

- Multiple source data capture - Allows processing of data from industry standard APIs and from any custom data interfaces that a particular application uses.
- Local persistence checking - Links corrective actions or responses to the repeated occurrences of a problem condition, so that a single point-in-time threshold exception does not immediately trigger a costly and unnecessary troubleshooting response. That is, it is important to ensure that individual events or conditions are not monitored in isolation, but rather within context of other related events around the same time period. For instance, if a monitor senses that CPU utilization is over 80 percent for one monitoring cycle, that could reflect a momentary spike and not an on-going condition. But if over a period of time it is recognized that the CPU is consistently utilized at over 80 percent, then this could indicate a condition that should be analyzed.
- Local intelligent correlation - Recognizes a number of metrics in aggregate as a *problem signature* enabling root cause identification and response to problems rather than symptoms. For instance, the fact that a client cannot access a database server could indicate that the database server has failed. However, if at the same time it is recognized that a network router is down

and would prevent clients from accessing the server, then it may not make sense to spend resources looking into a possible problem with the server.

- Real-time monitoring and reporting - Provides a real-time *heart monitor* that determines whether the application environment and individual applications are functioning properly.

1.4.5 Complex analysis

Autonomic managers need to have the capability to perform complex data analysis and reasoning on the information provided through sensors. The analysis will be influenced by stored knowledge data.

In April 2003, IBM introduced the autonomic computing blueprint, which is based on open standards and is designed to develop self-managing systems that use intelligent control loops to collect information from the system, make decisions, and adjust the system as necessary. The autonomic computing blueprint defines complex analysis technology building blocks that autonomic managers can use to represent knowledge, perform analysis, and do planning.

An autonomic manager's ability to quickly analyze and make sense of this data is crucial to its successful operation. Common data analysis tasks include classification, clustering of data to characterize complex states and detect similar situations, prediction of anticipated workload and throughput based on past experience, and reasoning for causal analysis, problem determination and optimization of resource configurations.

1.4.6 Policy-based management

An autonomic computing system requires a uniform method for defining the policies that govern the decision-making for autonomic managers. Policy specifies the criteria that an autonomic manager uses to accomplish a definite goal or course of action. Policies are a key part of the knowledge used by autonomic managers to make decisions, essentially controlling the planning portion of the autonomic manager. By defining policies in a standard way, they can be shared across autonomic managers to enable entire systems to be managed by a common set of policies.

1.4.7 Heterogeneous workload management

Heterogeneous workload management includes the capability to instrument system components uniformly to manage workflow through the system. Business workload management is a core technology that monitors end-to-end response times for transactions or segments of transactions, rather than at the component level, across the heterogeneous infrastructure.

1.5 Summary

This chapter has provided a brief overview of autonomic computing and the various capabilities that it encompasses. The Solution Installation capability is the focus of this publication. As enterprises strive to become more flexible and take advantage of software technologies to allow new solutions to be composed from existing components, the ability to be able to respond to new requirements and quickly deploy new solutions is critical.

Solution Installation defines standards and capabilities to allow software to be deployed in a manner that will minimize disruptions due to incompatibilities and missing components, while reducing the resources required to perform and track the deployment.

In the next chapter, we provide a more detailed overview of Solution Installation and its primary components.

2

Introduction to Solution Installation

Before we start describing the Solution Installation capabilities in detail, let us first look at some of the factors that are driving the requirements for it.

IT departments supporting today's enterprises are facing many challenges and are being asked to:

- Manage service levels.
- Increase resource utilization.
- Reduce costs.
- Respond to business challenges quickly and with flexibility.
- Manage increasing amounts of risk.

To meet these challenges, they are having to undertake several initiatives. Some of these include:

- Automated provisioning of systems
- Improved patch management (deployment and tracking)
- Improved asset utilization
- Optimizing testing environments
- Server consolidation
- Providing capacity on demand
- Ensuring high availability across systems
- Rapid repurposing of systems to meet critical business needs

- Adopting utility computing models
- Capturing IT best practices and automating them

Several of these require attention to installation of complex solution environments. The Solution Installation component of the autonomic computing initiative includes technologies and facilities to address the complexities of managing the installation of software components in a variety of environments.

2.1 Software install complexities

When looking at the process of installing complex solutions, there are many challenges and issues that arise.

In the past, a single solution might be installed on a single, stand-alone system. Though the installation with all of the prerequisite software may have been complex, it was handled by a specialist familiar with the application and all of its components. Once installed, the environment tended to be fairly stable and changes to it were made cautiously and through close adherence to change management processes.

But to meet the efficiencies required by today's business requirements, a single system may be running components used by multiple solutions; and a single solution typically depends on a variety of software components across multiple systems. In addition, in order to handle capacity requirements on demand, new systems need to be provisioned or reprovisioned with the appropriate components in a fast and efficient manner. In an on demand environment where responsiveness is key, systems may need to be repurposed to meet dynamic business requirements.

Manually installing and tracking all of the components associated with a solution and cross checking that with other co-resident applications for other solutions is a daunting and time consuming task for any administrator.

To build an environment where systems can be deployed, or redeployed, quickly and effectively, there needs to be advances in how solutions are deployed and tracked.

For instance, think about multiple solutions made up of multiple applications, each requiring components such as Web servers, application servers, databases and middleware that allows them to communicate with each other. Now add in version and patch dependencies, and the planning for what needs to be installed where and at what level is a major task.

Now think about the actual deployment. Different software components will have different installation programs with different levels of requirements for user interaction, and again various methods of prerequisite and co-requisite checking. What happens when a component such as the database engine needs to be upgraded to a new revision level to satisfy the requirements of one application? How will that affect other applications that may be using the same database engine?

Today, Solution Installations are carried out by administrators through a mix of methods, some manual and some automated. For a fairly static environment with

well defined operating platforms, most administrators have found ways to automate the physical installation. However, as the IT environment becomes more dynamic and deployment cycles are required to be significantly reduced, the ability for administrators to keep up with the business demands will be strained.

Problems include incomplete and inconsistent product information (such as prerequisites, capacity requirements, and so on), poor information regarding interoperability and support with differing versions of coresident software (operating system and other software), and poor tracking ability to help the administrator track what is installed (and how a new installation or upgrade affects other installed components).

The above issues cause installation and configuration failures, and significantly increase the cost of deployment. They also add costs to support/help desks and contribute to a loss of confidence in the supporting organizations.

2.2 Solution Installation basics

The Solution Installation component of the autonomic computing initiative is designed to put into place the standards, facilities and tools to enable administrators to handle the requirements around rapid and safe solution deployments.

Note: The latest version of the Solution Installation for Autonomic Computing toolkit is available in the Autonomic Computing section of the IBM alphaWorks® Web site (`http://www.ibm.com/alphaworks`).

The idea behind the Solution Installation technology/architecture is not to create a single installation program or technology, but to:

- Standardize the XML definition of an installable software component. This includes the definitions for dependencies, how to access the files to be installed, configuration information, relationship information between different software packages, and the actions to be performed during the operation.
- Provide a run-time environment that can consume these XML files and validate the target environment and installation plan before making any changes to the environment.
- Protect a working environment by checking how a new installation will affect currently installed components/solutions.
- Provide a run-time environment that will interface to hosting environments through a Web services interface to have the actions associated with the

change management operations carried out. (Recall that a hosting environment is a managed resource that can host software components.)

- Use the standard Web services interfaces for hosting environment touchpoints.
- Provide a registry and appropriate interfaces for logging and tracking what software is installed across an environment along with interdependencies and relationship information.

In simple terms, the Solution Installation provides:

- A standard way of defining installable units and all of their dependencies through XML, configuration information, actions and so on
- Standard run-time support that performs dependency checking and carries out change management operations
- Hosting environment support through touchpoints implemented as Web services
- An installation repository that tracks what is installed, along with complex relationship information

2.2.1 Concepts and components

To understand the overall structure and function provided by the Solution Installation, there are some basic concepts and components that must be understood. We provide an overview of these in this and the following sections, but leave the details for the rest of the book.

First a few key terms.

Installable unit

A software entity that can be deployed to an IT system. This entity could be simple, such as a single module or configuration change, or complex, such as multiple applications across multiple systems.

Smallest installable unit

A smallest installable unit (SIU) is a single entity. Think of it as an atomic unit of a software solution. For instance, it is often a single software application that has a single installation process. It is a single entity that, when installed, provides some meaningful function or information.

For example, a JRE might include a set of modules required at run-time and also a set of

documentation files. Each of these (documentation and run-time) might be represented by an SIU. So you can think of the SIU as being a minimal entity that provides some value when installed.

Container installable unit A container installable unit (CIU) is an aggregated installable unit. It could be made up of multiple SIUs or even other CIUs. It is intended to be installed to a single target host environment. For example, a CIU could consist of two SIUs, the JRE run-time and documentation as described above. When installing the CIU, each individual SIU could be installed as defined by the SIU, along with any specific options as specified in the CIU.

Solution module A solution module (SMD) is an installable unit that aggregates multiple SIUs, CIUs or other solution modules, typically targeted across a set of hosting environments. For instance, a simple component that makes up one piece of an application may well have files to be installed into the OS and an EAR file to be installed into WAS. This component might be represented by an SMD within a root IU.

Root installable unit The smallest unit of packaging. Root IUs have features, but SMDs, CIUs, and SIUs do not. A single solution is likely to be made up of multiple packages (root IUs) each containing multiple SMDs, CIUs and SIUs.

Important: Though the XML schemas exist for the definition of Solution Modules today, the current run-time environment provided by the Solution Installation toolkit does not yet support the installation of Solution Modules across different hosting environments on different machines. This support will be provided in a future version of Solution Installation.

The various installable units, as described above, provide the basis for the Solution Installation capabilities. We will describe how to create installable unit packages throughout this book. However, for now you should know that in general terms there are two groups of files that make up a package.

Payload files These are the actual files that make up the software product or solution to be installed. They are the

executables, libraries, jar files, help files and so on that need to be installed into the hosting environment.

Descriptors These are the XML files referenced earlier that describe the installable unit, the files it contains, the actions to be carried out for various change management operations, the prerequisites that must be met before installing the package such as capacity, operating environment and other required packages. Much more detail will be provided about these descriptors in below and in Chapter 3, “Descriptors and artifacts” on page 33.

The run-time components of Solution Installation will then parse the descriptors based on the requested operation and other information such as variables, options and the target environment, and manage the change operation. This includes performing dependency checks, putting together and executing an installation plan, monitoring the status of the various installations and updating a repository that tracks what is installed and the relationships between the installed components.

The architecture of the installable units and their associated descriptors provides a hierarchical way to manage the deployment of complex solutions.

A solution is any combination of products, components or application artifacts addressing a particular user requirement. This includes what would traditionally be referred to as a product offering (for example, a database product), as well as a solution offering (for example, a business integration platform comprising multiple integrated products), or a user application (such as a set of application artifacts like J2EE applications and database definitions). All the software constituents of a solution can be represented by a single Installable Unit Deployment Descriptor (IUDD), described below, as a hierarchy of installable unit aggregates. The top-level aggregation is the root installable unit. In addition to the installable units that comprise a solution, the IUDD also describes the logical topology of targets onto which the solution can be deployed.

2.2.2 Descriptors

Information about installable units is stored in XML documents called deployment descriptors. The deployment descriptors provide a standard way to define an installable unit, and information that is required for its installation. This might include information regarding dependencies on the environment (such as required disk space, processor, operating system, and so on) and dependencies on other software packages (co-requisites, prerequisites, and so on).

Other information such as configuration options, the location of the files to be installed, and actions to be taken for a particular change management operation

(install, upgrade, uninstall, and so on) is also made available either in the deployment descriptor or, more typically, in other files referenced by the deployment descriptor (such as media descriptor files discussed later).

The various files and types of information that are described in them are consumed by different components within the Solution Installation framework. For instance, information about dependencies is used by the DependencyChecker component described shortly, while information about the actions to perform for a specific change management operation would be utilized by the touchpoint associated with the managed resource.

This will all become clearer as we describe more of the base components and the descriptions of the descriptor files throughout this book.

The run-time environment will utilize the deployment descriptor (and other referenced files) to check dependencies, validate that the installable unit should be and can be installed, and what should be done for various change management operations. The deployment descriptor becomes the definitive reference regarding how an installable unit should be handled. Note, however, that options such as selectable features and target installation directories can be overridden at installation time. But any such options or variables are still defined within the descriptor files.

Chapter 3, “Descriptors and artifacts” on page 33, provides details on deployment descriptors and other related files.

2.2.3 Change manager

The change manager is a central part of the Solution Installation run-time environment. It takes the deployment descriptor as input and coordinates the change management operations across the appropriate hosting environments. It utilizes other run-time components such as the dependency checker and communicates with the registry and touchpoints. The primary interface for installation programs is the change manager.

The change manager does not perform the actual actions of a change management operation. Based on the requested change management operation (create, upgrade, delete, as examples), the descriptor files will define the actual actions to be carried out. The change manager passes this information on to the touchpoint associated with the target hosting environment and those actions will be carried out by the touchpoint. However, the change manager does coordinate these operations and tracks their progress and success. If an operation fails, it is the responsibility of the change manager to then initiate activities such as backing out the necessary changes to put the system into a known and stable state.

In general, the change manager provides the logic to:

- Determine what installable units must have change management operations applied, based on information in the descriptors and information passed from the installation program (including options specified by the user).
- Validate that all dependencies of the target environment are met by utilizing another run-time entity called the dependency checker.
- Determine what order operations on various installable units should be applied.
- Execute the change across the target hosting environments by communicating with the touchpoints.
- Register the resulting changes with the installation database (registry).

Installation programs, or systems management programs, can interface directly to the change manager. These calls are asynchronous due to the fact that some change management operations can be time consuming and may even involve the reboot of a system.

The typical interactions with the change manager might include the following:

- Submit a change request.
- Receive a notification of change.
- Perform a status check on a pending change.
- Cancel a pending change.
- Resume a suspended change request/operation.

Processing performed by the change manager can include the handling of features and installation groups. We will define these in more detail later, but for now, features support the installation of optional components of an installable unit. Feature processing can be quite complex because it introduces new levels of interdependencies. Install groups are mechanisms for grouping commonly associated features. For instance, by choosing to install a specific install group, a predefined set of features would be selected.

2.2.4 Dependency checker

The dependency checker is a component of the run-time environment of the Solution Installation. An installation or other systems management program can call the dependency checker directly to validate the installability of a particular installable unit. Otherwise, an installation program could leave it up to the change manager to interface with the dependency checker.

For a simple environment, the installation program might just allow the change manager to handle the dependency checking. In this case, if the change manager determines (with the help of the dependency checker) that a solution

cannot be installed, then it would typically return back to the user with such an indication.

However, if the creator of the installation program wanted to provide more robustness, the dependency checker could be used directly. Based on the results of the check, the installation program might choose only those features or install groups that are legitimate for the given environment. Further interaction with the user could be involved to explain the options and allow the user to choose the correct course of action.

Related to, but not to be confused with, dependency checks are entities called elementary checks. An elementary check simply provides information. For example, someone might include an elementary check that checks a certain environment condition. The results of that check might be used to invoke a conditional action or as input to an action even though the check does not represent a dependency.

Elementary checks can include items such as the target operating system amount of disk space or processor speed.

Dependency checks can also involve more complex checking such as prerequisites (a component is needed before the other is installed), co-requisites (a component is needed in parallel with another), or ex-requisites (a component must not be installed in the same environment as the candidate installable unit). (Some of these capabilities may not be available in the current release.)

2.2.5 Installation database (Registry)

One of the complexities associated with installing software solutions is the tracking of what is already installed and the inter-dependencies between installed components. Solution Installation provides an installation database or registry and interfaces for registering installed software along with information such as relationships between installed software. The dependency checker can utilize the registry to determine that requirements for a particular package can be met.

The registry not only provides performance benefits compared to having to dynamically scan a potential target system, but it also maintains complex relationship information as defined in the descriptors for all of the packages that have been installed. Therefore, the information in the registry will allow the change manager to protect the overall environment. For example, if a request is made to install an upgrade to DB2, the registry will contain information about other installed programs that also have a dependency on DB2 and would be able to indicate to the change manager that this upgrade might break another application that is not supported on the latest version of DB2.

2.2.6 Touchpoints

The Autonomic Computing architecture differentiates between managing systems and managed systems. Managed systems are controlled through an externalized interface called a touchpoint. The touchpoint is implemented as a Web service and provides a well defined set of interfaces that management applications can utilize to manage the resource represented by the touchpoint.

The interfaces to touchpoints are defined using WSDL. Currently, their are two touchpoint types supported by Solution Installation. There is a touchpoint for operating systems and a touchpoint for J2EE containers such as WebSphere. Each type of touchpoint supports a set of actions appropriate for the hosting environment. For instance, the operating system touchpoint supports actions such as creating directories, creating/copying files, executing external commands and so on.

The change manager will pass information to the touchpoint (such as the location of the source files and the set of actions to perform). The touchpoint will then carry out the actions required for the change management operation.

2.2.7 Installation program

The primary installation program in a Solution Installation environment is responsible for interfacing with the user and the change manager on the management server. This would include passing the deployment descriptors to the change manager along with any user definable information (such as feature selection, configuration options, the change management operation to be performed, and so on). The change manager and the rest of the Solution Installation runtime objects then validate (and in some cases choose) the target environment and use the touchpoint to perform the actual installation as defined by the deployment descriptor and other artifacts. The change manager object is also responsible for updating the registry with information about the installation and calling back to the installation program with the overall result.

One often asked question is regarding currently existing installation programs for specific software packages. Do they need to be rewritten to take advantage of Solution Installation? It is important to remember that Solution Installation is intended to coordinate the installation of solutions that are typically made up of multiple components and possibly across multiple hosting environments.

So the installation program we are discussing in this section is the coordinating program that takes advantage of the Solution Installation run-time and facilities.

But if you have a software package with its own installation program that can run in unattended mode (such as an InstallShield or ZeroG module), then this can still be used by the touchpoint to perform the physical installation. In this case,

the descriptor files would define the actions (such as launching the InstallShield or ZeroG installer) along with the physical location of the product-specific installation programs. The change manager would pass this information to the touchpoint that would be responsible for accessing and launching the product-specific installation module.

So the answer to the question is that individual, currently existing installation programs that can be run in unattended mode can generally be called by the touchpoint as the result of a change management operation request.

2.3 Flow and relationship between components

Up to this point we have described the major concepts and components of the Solution Installation. Let us take a look at how these components relate to one another and interact.

The following figure provides us with a reference for our discussion.

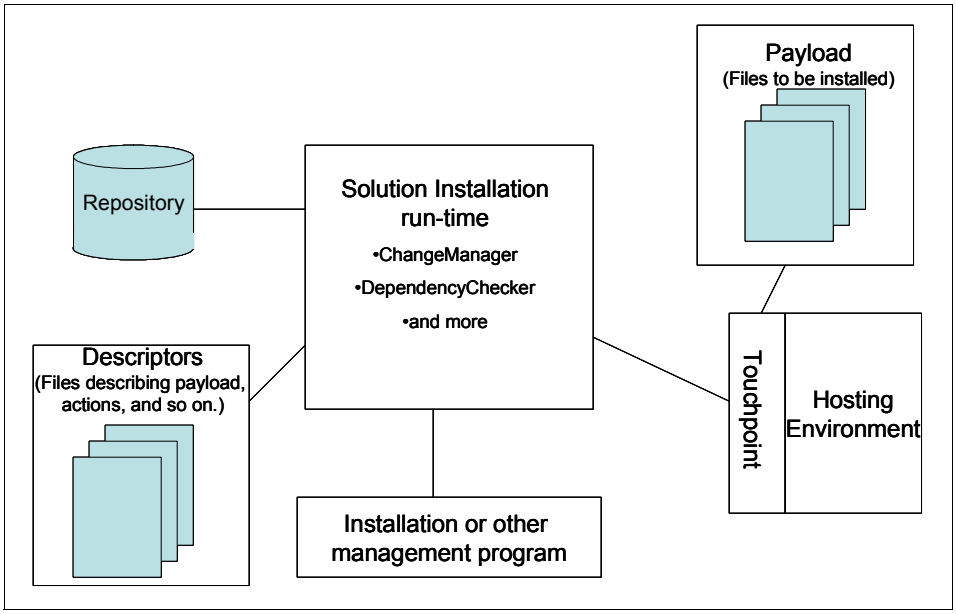

Figure 2-1 Components of Solution Installation

Though the Solution Installation supports a wide range of change management operations, for this discussion let us assume we are talking about an installation operation. The files that need to be copied to the target hosting environment are called the payload and are represented in the upper right of the figure. The

descriptors that describe these files, the desired target environments, the requirements that must be met before an installation can occur, the location of the files, and the actions that should be carried out as part of the change management operation are shown in the lower left.

An installation or other management program (shown in the bottom of the figure) is responsible for interfacing with the Solution Installation run time made up of objects such as the ChangeManager and DependencyChecker. The installation program (using other objects/facilities provided with Solution Installation) is basically responsible for passing the information contained in the descriptors to the ChangeManager. The ChangeManager will utilize the DependencyChecker to validate the target environment by using information in the repository and by communicating with the target systems through their touchpoint interface.

Assuming everything is in order, the ChangeManager will then ask the touchpoint to perform the actions associated with the change management operation. Once complete, the ChangeManager will register the appropriate information about the newly installed software with the repository and notify the installation program of the completion of the change management operation.

2.4 Who can/should use it and how

Now that you have a high level view of what the Solution Installation is all about and before we get into more of the technical details, you may be wondering who the intended user of this technology and toolkit is.

There are several types of people and organizations that would benefit from the use of the Solution Installation. Some of these are highlighted below:

- Software vendors

  Software vendors can benefit from this technology as it provides a standard way for them to define the actions required for various change management operations for their software. By using XML-based descriptor files to define the installation requirements, options, actions and so on for their products, their products will be more easily and cost effectively managed and therefore more attractive to their potential customers. Depending on their product line, it will allow them to create a hierarchy of installable units that can reference one another and whose relationships with one another can be well-defined in an industry standard format.

  Their products will be easier to integrate into complex solutions being built by large enterprises that are looking for software that can be easily deployed and managed through the installation life cycle.

Aside from creating the appropriate descriptors for their products, software vendors can develop installation programs that will take advantage of the Solution Installation run-time environment and touchpoints. By doing so, they can spend less resource writing installation logic and more resource on their product's unique capabilities.

- Installation program and systems management vendors

  Companies who specialize in developing installation software or other systems management software handling change management can now settle on an open, standards-based architecture. The richness of the XML schema definitions for the various descriptor files provides the capability for installation and change management software to write more generic code for handling the descriptors and identifying the options that are made available to the user (through a GUI or response files). Installation programs can simply interface with the change manager as described earlier or take advantage of other components of the run-time such as the dependency checker to provide a rich set of capabilities.

- Enterprise IT departments

  Enterprise IT departments need to find ways to more efficiently deploy complex solutions across a distributed environment. Performing this task today is tedious, time consuming, expensive and susceptible to errors due to inconsistent installation programs, and ways of handling change management operations.

  These departments will be able to take advantage of more functional installation programs provided by their software vendors based on the Solution Installation technologies.

  Some enterprises may also want to assemble complex solutions out of various software products from a variety of vendors. Today, some enterprises develop their own deployment frameworks, but are looking for a unified approach to handling the installation (and other change management operations) of these solutions. Having a consistent, open, standards-based method of defining installable units (SIUs, CIUs, and Solution Modules) will make it much easier and more cost effective to deploy their business-critical applications in a timely manner.

  Other change management operations such as patch management may also be much easier when using Solution Installation technologies.

Is this a mature set of capabilities that are widely available across all platforms today? No, not yet. But because of the openness of the technologies and the use of widely accepted standards such as XML, Java and Web services, developers across all organizations can start taking advantage of the architecture and tools that are available today.

IBM will certainly be taking advantage of this technology across all of its software brands to make its products more manageable and easier to compose into complex solutions.

2.5 Solution Installation contents

We have described some of the basic technology components earlier in this chapter. In this section, we provide a brief overview of what is actually included in Solution Installation Version 1.2 and is available as of the release of this book.

2.5.1 Obtaining the toolkit

The current version of the Solution Installation toolkit as described in this publication is V1.2. It is available from the IBM alphaWorks Web site using the following URL:

http://www.alphaworks.ibm.com/autonomic

2.5.2 Toolkit contents

The toolkit that can be obtained as described above includes the following components.

- Run-time environment
- ManageIU sample installer
- Registry utilities
- XSD files
- WSDL files
- Samples
- Documentation
- Operating system touchpoint

Restriction: Note that the current version of the toolkit is not a complete implementation of the architecture. For instance, the current touchpoint bindings do not support SOAP/HTTP, but rather a local Java binding, limiting the change management operations to the local system; that is, the Solution Installation run-time will execute on the managed system and access the touchpoint through local Java calls. These restrictions will be lifted in the future. This version of the toolkit allows developers to start gaining experience with the toolkit and the Solution Installation architecture.

When the toolkit is installed in a Windows® environment, the default location is \Program Files\IBM\common\acu. In a Unix or Linux environment, the default

location is /usr/ibm/common/acu (with some ‘fixed’ files placed in /var/ibm/common/acu).

In the following discussion we will refer to the above paths generically as <installdir>.

Within <installdir> there are several directories that will be briefly described below. There are also setenv.cmd and setenv.sh shell scripts to be used to set the appropriate environment variables for execution of the Solution Installation run-time commands. More on these commands shortly.

Under <installdir> the following directories are created:

./bin	Primarily includes various command files or shell scripts
./dat	Contains directories and data related to the Solution Installation collectors (used when collecting and storing information about target environments). It also contains a wstp directory containing files related to the WebSphere touchpoint.
./lib	Primarily contains jar files and other libraries needed for the execution of various commands and run-time components.
./logs	A directory for the writing of log files generated by the Solution Installation run-time.
./repos	Holds the installation database or repository.
./schema	Contains the schema definition (.xsd) files for the various XML descriptors and artifacts used by Solution Installation.
./wsdl	Contains the WSDL and other supporting files that define the touchpoint interfaces as Web services and their associated bindings.

Within the ./bin directory, there are several commands that can be used to explore and utilize the Solution Installation facilities. To use these commands, ensure the environment is properly set by executing the **`setenv`** command in the root of the <installdir>. The commands available include:

backupdb	A script that backs up the Solution Installation registry.
exportIU	Exports the deployment descriptor for an IU from the installation registry.
listIU	Lists the installed IUs per the installation registry

manageIU	A sample program that can be used to invoke change management operations. This program is not meant to be used in a production environment.
registerIUDD	Can be used to register an IU in the installation registry by referencing the deployment descriptor file.
restoredb	Restores a previously backed up version of the Solution Installation registry.
validateIUDD	Validates an IU deployment descriptor before using it to perform a change management operation.

2.5.3 Current capabilities and limitations

It should be restated here that this publication applies to our experiences with an early release of Version 1.2 of the Solution Installation toolkit. Some of the content described throughout this book applies to the Solution Installation architecture and may or may not be fully implemented in the current version of the toolkit.

The toolkit will be enhanced over time and capabilities will be implemented and added. The toolkit documentation available with the current version of the toolkit will provide specifics as to what is and is not currently available and implemented.

2.6 Summary

This chapter has provided a brief overview of the Solution Installation concepts and basic components. Throughout the rest of this book, we describe specific components in more detail and provide examples that you can take advantage of to start learning more about the capabilities and how they can be used to meet your business needs.

3

Descriptors and artifacts

This chapter covers information related to deployment descriptors and other artifacts that provide a standard way of describing the solution to be installed and all of its characteristics. Though we have already mentioned the deployment descriptor, there are other descriptors that describe the media that contains the files to be installed, configuration information, and actions to be performed. The use of these files and their schema definitions are described in this chapter.

Note: The examples and descriptions in this chapter are based on our work with an early version of the Solution Installation for Autonomic Computing toolkit. Therefore, some differences may exist with the generally available version.

3.1 Introduction to deployment descriptors

A deployment descriptor is an XML document that describes the install characteristics of Installable Unit (IU) packages. It describes the capabilities and dependencies of the package. A deployment descriptor is provided as input to other components in the Solution Installation framework.

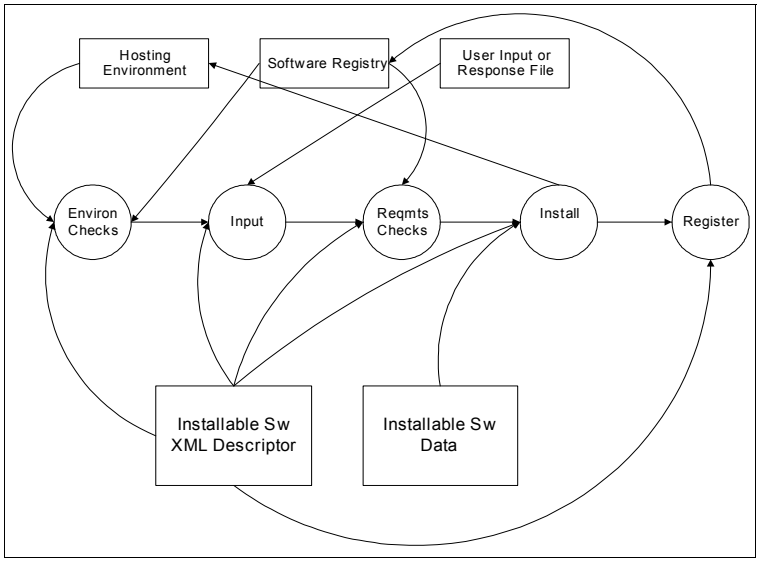

Figure 3-1 Data flow diagram of software installation process

Figure 3-1 provides a model of the software installation process as a sequence of five logical activities: Environment checks, user input, dependency (or requirement) checks, installation and registration. In all five activities, the deployment descriptor can be used as an input provider.

The deployment descriptor is an XML file whose contents are defined by a series of schema definitions (.xsd files) provided with the Solution Installation toolkit. These schema files define the grammar and syntax of the various elements of a deployment descriptor.

Once the Solution Installation toolkit is installed, the schema definition files are located in the %INSTALLDIR%/ACU/schema directory. The default installation directory on Windows is c:\Program Files\IBM\Common.

3.1.1 Schema file overview

The schema files that define the valid content of a deployment descriptor include:

- base.xsd - Defines the basic features of a smallest Installable Unit. The elements defined include the grammar and syntax of:
  - Display items - Information used by a user interface to display information about the SIU
  - Identifiers and expressions - Used within the deployment descriptor to identify the SIU
  - Property names
  - Variables
  - Identity information - Used to provide information about the specific SIU and its type (base install, fix, and so on)
  - Checks - Specification of environmental factors such as capacity, consumption, system version levels and so on
  - Life cycle (change management) operations - How to specify phases of the life cycle such as create, update, undo, initial configuration, migration and so on
  - Other miscellaneous items
- action.xsd - Defines action groups, base actions, and variables related to actions that will be performed for various change management operations.
- command.xsd - Defines the grammar and syntax associated with commands to be executed as part of a deployment. This includes information about the location of the commands, types of commands (external programs or Java classes), how to handle standard and error output, checks on the environment before executing a command, and so on.
- config.xsd - Defined information used for configuration, such as property values.
- iudd.xsd - Defines the grammar and syntax used when aggregating or referencing other installable units.
- osActions.xsd - The elements defined in this schema file are derived from action.xsd and define the syntax and grammar to use for defining actions to be performed by an operating system touchpoint.
- relationships.xsd - Defines the elements that specify various relationships among components of a solution such as uses, hosts, deploys, supersedes, and several others.

- resourceTypes.xsd - Enumerates various resource types. These resource types are currently categorized as OS resources, J2EE resources, or relational database resources.
- signatures.xsd - Defines various signatures to be used to identify components. These signatures could be combinations such as file name, file size, check sum, and so on, or could include OS-specific registry (such as the Windows registry) information. Deployment descriptors could use various signature definitions to identify requisite products, for instance.
- siu.xsd - Defines the grammar and syntax related to SIUs including references to other artifacts such as configuration units, custom checks, requirements, and so on.
- version.xsd - Defines the syntax for specifying version strings.

If you are familiar with XML schema definitions, the best reference for the above .xsd files are the files themselves.

3.1.2 Deployment descriptor structure

The structure of a deployment descriptor is represented in the figure below.

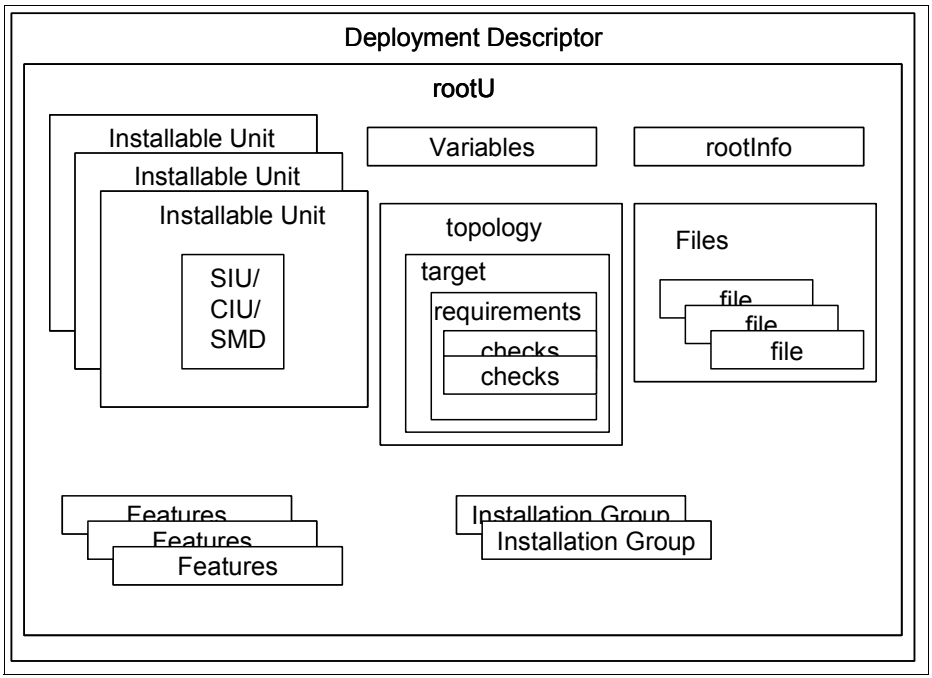

Figure 3-2 Block diagram of the contents of a deployment descriptor

Every deployment descriptor is a root installable unit. This root installable unit may contain or reference other installable units within its definition. The deployment descriptor for the root installable unit is defined by the rootIU element. Each deployment descriptor contains one and only one rootIU <rootIU> element. The rootinfo element contains information about the deployment descriptor and the schema against which the descriptor is created. The deployment descriptor has one or more installable unit elements with one or more feature and installation groups, a topology element, and a files element that in turn has one or more file elements.

3.1.3 Root installable unit

The root installable unit, defined as <rootIU>, is an AggregatedInstallableUnit type. That is, it can contain more than one installable unit, which, in turn, is an InstallableUnit type. The installable units contained in a root installable unit can be smallest installable units, container installable units, or solution modules. A rootIU element defines the following information:

- Schema files used
- Location of these schema files
- Name of the rootIU

Example 3-1 rootIU element

```
<iudd:rootIU
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:OSRT="http://www.ibm.com/namespaces/autonomic/OS_componentTypes"
xmlns:J2EERT="http://www.ibm.com/namespaces/autonomic/J2EE_RT"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUD
D iudd.xsd"
IUName="HelloWorldCC">
```

Every installable unit (except those describing fixes) has its own identity element within the deployment descriptor. The rootIU element also has an identity element, since it is an installable unit itself.

Note: For all IUs (including rootIU) there are two names associated with it. One in the IU declaration, as shown in the above example. Another in the IU's identity element, as shown below.

The IUName is only used as an internal identifier within a particular descriptor. The name that is registered in the software registry is the identity name described in 3.1.5, "IUDefinition" on page 40.

RootIU Identity

RootIU identity is an element that describes the rootIU element. It includes the rootIU name, the Universally Unique Identifier (UUID) (the UUID is described in 3.1.4, “Installable unit elements” on page 38), and version of the software. It also has optional buildID and buildDate elements for reference.

Example 3-2 The rootIU identity element

```
<identity>
   <name>Hello World Common Component</name>
   <UUID>9ac214510ba426fe1f5384140235cceb</UUID>
   <version>1.1.0</version>
</identity>
```

3.1.4 Installable unit elements

The installableUnit element defines the type of installable unit. Installable units are one of the following types:

- Smallest installable unit
- Container installable unit
- Solution module
- Configuration unit (This and some other types not listed are not supported in the current toolkit.)

Although a configuration unit is an installable unit it is quite different from the other IU types and is applied after an install/uninstall. Configuration units are discussed in 3.3.2, “Configuration artifacts” on page 74, of this chapter.

In addition to defining each installable unit, the attributes for the installableUnit element provide the following information:

- A target reference that refers to one of the targets defined in the topology element (see 3.1.8, “Topology element” on page 53, for details). Target references are optional for RootIUs or solution modules. They are required for all CIUs or SIUs if not already specified by its containing aggregated installable unit.
- The conditions that identify whether the installable unit should be installed

Example 3-3 Simple installableUnit element example

```
<installableUnit targetRef="tOperatingSystem">
   .
   .
   .
</installable Unit>
```

Smallest installable unit

This element describes the smallest installable unit of software that can be deployed to a single hosting environment. This element is defined as <SIU> in the deployment descriptor.

Example 3-4 Declaration of an SIU element

```
<installableUnit targetRef="tOperatingSystem">
   <SIU IUName="HelloWorldIU" hostingEnvType="OSRT:Operating_System">
      .
      .
      .
   </SIU>
</installable Unit>
```

Container installable unit

A container installable unit (CIU), defined as <CIU>, is a type of aggregated installable unit that is targeted at a single hosting environment. It contains other SIUs and CIUs, which are either defined inside the deployment descriptor or an external software package whose reference is inside the deployment descriptor. The inclusion of these IUs can be conditional based on a variable expression placed inside the CIU definition element.

A container installable unit has the following fields:

- The identity of the installable unit
- Constraints on whether the installable unit can be shared across different software packages
- The checks to be performed and requirements on that hosting environment
- A mandatory IUname attribute that is used to reference the installable unit

Example 3-5 CIU element declaration

```
<installableUnit targetRef="tOperatingSystem">
   <CIU IUName="HelloWorldCIUWrapper" hostingEnvType="OSRT:Operating_System">
      .
      .
      .
   </CIU>
</installable Unit>
```

Solution module

A solution module is a type of aggregated installable unit. It should be a multi-target installable unit. The contained installable units can be smallest installable units, container installable units, or solution modules.

The main elements are the:

- Identity of the solution module.
- Variables, including parameters that may be used to customize the solution module.
- Contents, which may be a combination of single target installable units or of other aggregated IUs.
- An optional condition that identifies whether the installable unit should be installed. The default is true.

The following example shows a portion of a simple solution module definition.

Example 3-6 Solution Module declaration element containing an SIU

```
<solutionModule IUName="HelloWorldSolutionModuleWrapper">
   <identity>
      <name>Hello World Wrapper</name>
      <UUID>db6c9c23b63f5af0f26dc819cd9a2786</UUID>
      <version>1.0</version>
   </identity>
   <installableUnit targetRef="tOperatingSystem">
      <SIU IUName="HelloWorldIU" hostingEnvType="OSRT:Operating_System">
         .
         .
         .
</solutionModule>
```

The schema elements *IUDefinition*, *variables* are common to an SIU, CIU or solution module. The unit element is specific to an SIU. Each of these elements are defined in the following sections.

3.1.5 IUDefinition

An installable unit definition contains information about the identity, signatures, checks, requirements, constraints and consolidated fixes of the installable unit. These are key data used to populate or to query the software registry. All dependencies declared within the IUdefinition element are checked against the target container (property checks) and the software registry (software checks).

The information in the IU definition is structured as illustrated by the following diagram.

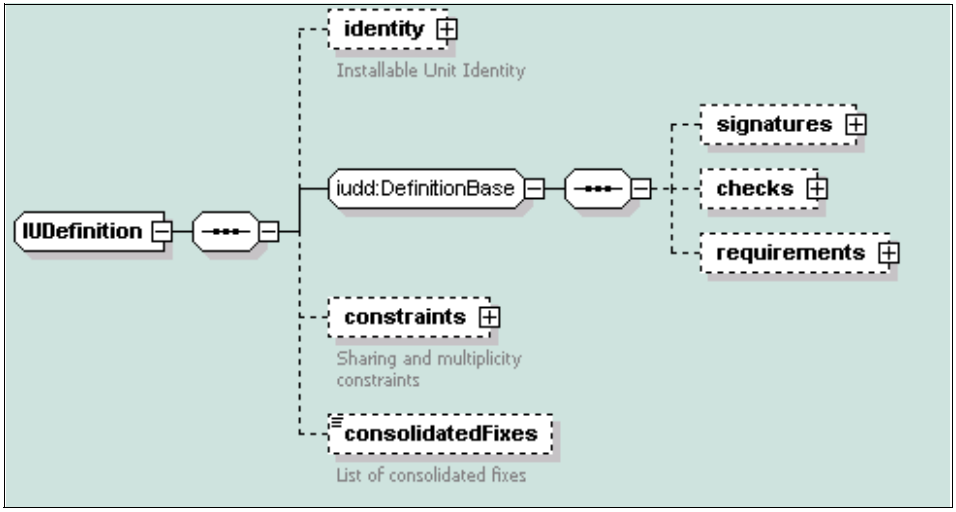

Figure 3-3 Structure of an IUDefinition

The content of each element in the above diagram is described below.

Identity element

The identity element, <identity>, is a mandatory element for an installable unit whose purpose is to identify an installable unit. The elements within the identity element are:

- name - This is an internal name of the component. This element is provided for the following non-exclusive uses:
  - The IU may be installed on a system and be registered in a legacy registry of installed software by some printable name. This element could specify the same printable name in the IU XML descriptor, thus making it possible to identify an instance found in the legacy registry as an instance of this SIU.
  - A system providing life-cycle management of installable units, like a library system, needs to provide a means by which an installable unit can be referred to by a name that is easy to remember. The UUID is not suited for that purpose. As an example: When creating a new version of an existing application, a developer may need to retrieve the UUID of the base version. The base application can be retrieved by searching a library system by this name.
  - One may need to search for different products that have different UUIDs yet are all members of a product family. For example, assuming SomeProduct version Y does not declare backward compatibility with SomeProduct version X, the two must have different UUIDs. Still, an

application may want to find all instances of the SomeProduct family installed on a given computer. Having both SomeProduct X and SomeProduct Y registered with a name that contains the “Someproduct” sub-string would make this correlation possible.

- UUID - This is a unique, universal identifier of the component. Every IU in the universe has its own UUID value created by using the MAC-address, date, and time of the computer system so that no two IUs may be accidentally assigned the same UUID. A new version of an existing component that maintains compatibility to prior versions of the same component must carry the same UUID. A new UUID should be created when the component is first created, when a new version is released that breaks backward compatibility or in case the component is broken apart and components of it are used to reconstruct something different. The UUID and the version of the installable unit are used to uniquely identify the associated installed unit.
- Version - The <version> element indicates the version of the IU. This element is an instance of the base:Version type, which includes four attributes. These attributes (ver, rel, mod and lev) define the version according to the “VRML” (version, release, modification, level) schema. Only the first two elements (version.release) are mandatory. The modification and level elements are optional. This element is only used in the context of the IU identity definition. A more compact declaration of version is provided by a different type (base:VersionString) used in some contexts.

Example 3-7 Sample of identity element for an IU

```
<identity>
      <name>Hello World</name>
      <UUID>db6c9c23b63f5af0f26dc819cd9a2787</UUID>
      <version>1.0</version>
</identity>
```

Signatures element

The signatures element, <signatures>, defines the signature of the software component being installed. This is an optional element in an IU definition. A component may be installed with a nonstandard installer that does not record its instance and type in the standard registries. An inventory scanner can use the component’s signature definitions to discover instances of the component in the hosting environment.

The signatures element includes an unbounded sequence of signature elements. This element defines only one optional attribute, keySignature, that is common to all signatures. A value of “true” indicates that the presence of the signature is a necessary condition for an instance of the IU to be considered installed.

Specifying more than one signature with the keySignature=”true” attribute implies that all of them must be present for the IU to be considered installed.

A signature with keySignature=false indicates an additional file whose existence should be checked for other purposes (for example, license management or integrity checking).

The specification of a signature depends on the IU hosting environment. One or more signature types may be defined for a hosting environment. The operating system hosting environment defines two such types:

- FileSignature: These elements define a key to be found in the software registry.
- WindowsRegistrySignature: These elements define a key to be found in the Windows registry.

Checks element

This is an optional element within the IU definition. This element can also be placed inside the target element.

Checks are actually a type of variable. They may be used in places other than the requirements sections. Other useful places include conditions for IUs, actions, action groups, artifact sets, and derived variables.

The checks element, if present in the IU definition, is used to define checks to be made on the target hosting environment or on the collocated operating system. A check returns the result in a boolean variable, which is defined as part of the check. References to the results of a check are made in the declaration of requirements that need to be met before the component can be installed.

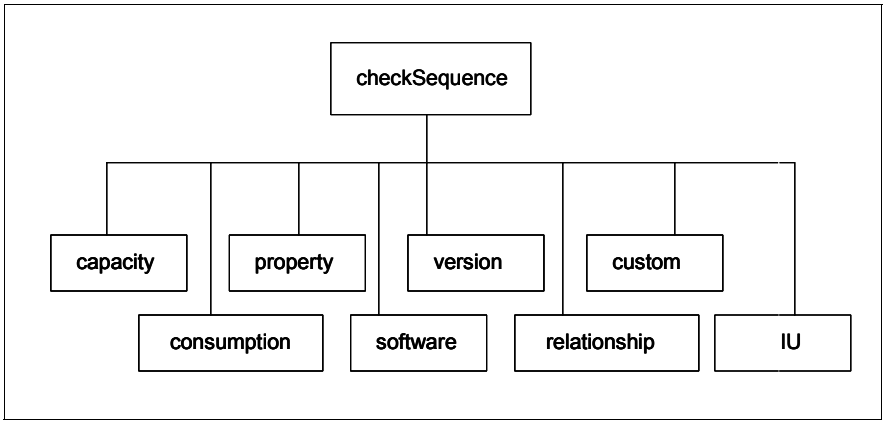

Figure 3-4 Check elements

The schema enforces a separation between checks and requirements. Checks may be used to obtain information about the target hosting environment and to condition the install behavior, like the execution of specific actions, upon their result. Requirements, as described in the following section, are used to formally express one or more alternatives, one of which at least must be satisfied in order for a component to be installable. For instance, a check on the OS version might alter the options or actions taken during an installation, where a requirement for an OS version would only allow the installation to take place if the target system is running the specified OS version.

Different types of checks are supported: Capacity, consumption, property, version, software, relationship, IU and custom.

The checkId attribute can be specified for all types of checks. It is used to associate a symbolic name to the check being defined. This name is used to refer to the result of the check in conditional expressions and requirements. This is a required attribute.

Requirement element

The requirement element, <requirement>, defines the requirement on the hosting environment that must be satisfied for an installation to be performed. Multiple requirements can be defined in the IU definition of a component. These requirements can be stated in terms of the elementary checks introduced above.

Each requirement element defines one or more alternatives. Each alternative defines a condition on the target hosting environment. At least one alternative must be satisfied for the requirement to be met. The failure to meet any requirement that is defined by a unit causes the unit installation to fail before any action is executed. A requirement element has two attributes, name and operations. The name attribute is the identifier of the requirement and is a required attribute. The attribute operation is optional, and can contain the value of any valid operation as defined in the base.xsd file. See Chapter 4, “Solution Installation operations” on page 77, for more information on change management operations.

Each alternative is defined as a sequence of elementary checks whose result must be verified. An elementary check is either referred to by the checkItem element or it can be defined in line within the inlineCheck element.

The alternative element has the attributes name and priority. Attribute name is the identifier of a variable containing the boolean result of the alternative. This variable is “true” when all the check items within the alternative are satisfied. This variable can be used in conditional expressions or to condition the setting of other variables. This is a required attribute. The attribute priority defines the priority amongst the alternatives, and is a non-negative integer. This attribute is

also optional. A zero value is the lowest priority, with higher numbers indicating higher priorities.

CheckItem and inlineCheck

An inlineCheck element can be used instead of a checkItem element to specify a check directly within an alternative element; that is, without making a reference to the result of an elementary check that is performed before the user input phase. It can be used to specify one of the checks of type: Capacity, consumption, property, version, IU, custom, relationship or software. A checkItem element within an alternative has the following attributes:

- checkIdRef - This is the name of the variable containing the boolean result of an elementary check.
- testValue - When this optional attribute is not specified the default is assumed and the check result must be the boolean “true” for the alternative to be satisfied. Specifying testValue=”false” means that the check result must be the boolean “false” for the alternative to be satisfied. In the case of a software check, testValue=”false” means that the specified software IU is an ex-requisite (another package should not be installed if the installation is to continue).

Example 3-8 An inlineCheck element

```
<requirement name="OS">
   <alternative name="Windows_2000_Check" priority="0">
      <inlineCheck>
         <property checkId="Windows_2000_Check">
         <propertyName>OSType</propertyName>
            <value>Windows 2000</value>
         </property>
      </inlineCheck>
   </alternative>
</requirement>
```

Adding requirements for specific operations

Some installable units may not need to declare requirements for any change management operation other than install. This is the case if it can be assumed that the execution of other operations like configure or uninstall should not require additional resources. In general, however, there may be extra consumption resources (such as temporary disk space, CPU speed, and so on) that are needed during any of the successive operations that may be performed during the unit’s life-cycle.

Requirements can be added that precisely list the checks that must be executed when executing one or more operations. In the current example, let us assume that there are actions, within this unit of software, executing a configuration

program during the SelfTest operation, and this program needs a particular CPU speed. A requirement can be added causing the following check to be executed (Example 3-9 on page 46).

Example 3-9 Adding a self test

```
<requirement name="ConfigureAndTest_Reqmt" operations="Configure">
   <alternative>
      <checkItem checkVarNameRef="OS_CPU_SpeedCheck" />
   </alternative>
</requirement>
```

Available checks

Below, the various checks that have currently been defined are described.

IU check

An IU check determines whether an IU is installed on the hosting environment. An IU check is defined with the IU element. The IU check queries information in the installation database of the hosting environment. IUs can be queried based on name, UUID, and include other attributes such as version numbers.

Example 3-10 IU check example

```
<iu checkId="PreReq">
   <name>Requirement Package</name>
   <minVersion>1.2</minVersion>
</iu>
```

Capacity check

This is used to check the value of a named property defining some capacity of the target hosting environment. WSDL files for each of the supported hosting environments define the capacity properties supported. As an example, the operating system hosting environment property “processor/MaxClockSpeed” holds the processor frequency in MegaHertz.

Example 3-11 Capacity check example

```
<capacity checkVarName="OS_CPU_SpeedCheck" type="minimum">
   <propertyName>MaxClockSpeed</propertyName>
   <value>2000</value>
</capacity>
```

Consumption check

This is used to check the availability of resources to be consumed on the hosting environment, such as disk space. The consumption requirement can be

temporary or permanent. As an example, the operating system hosting environment property FSFreeSize holds the available disk space for a particular file system.

Example 3-12 Example of a check for available disk space on a system

```
<requirement name="myReq3">
   <alternative name="myAlt1">
      <inlineCheck>
         <consumption checkId="consumptionCheck1" targetRef="tFileSystem">
            <propertyName>availableSpace</propertyName>
            <value>100</value>
         </consumption>
      </inlineCheck>
   </alternative>
</requirement>
```

Property check

Property check is used to check the value of properties defined for a hosting environment, in addition to capacity and consumption properties. As an example, the operating system hosting environment supports the property OSType.

Example 3-13 Property check element

```
<property checkVarName="Windows_2000_Check">
   <propertyName>OSType</propertyName>
   <value>Windows 2000</value>
</property>
<property checkVarName="Windows_XP_Check">
   <propertyName>OSType</propertyName>
   <value>Windows XP</value>
</property>
<property checkVarName="Linux_Check">
   <propertyName>OSType</propertyName>
   <pattern>.*LINUX.*</pattern>
</property>
```

Version check

Version is used to check the value of a property of a hosting environment that has the characteristics of a version. As an example, the operating system hosting environment supports the property “version”.

Example 3-14 Version check element

```
<version checkVarName="Windows_version_Check">
   <propertyName>version</propertyName>
   <minVersion>5.0.2.3</minVersion>
   <maxVersion>5.0.3.0</maxVersion>
```

```
</version>
<version checkVarName="Linux_version_Check">
   <propertyName>os_version</propertyName>
   <minVersion>7.2</minVersion>
   <maxVersion>8.1</maxVersion>
</version>
```

Software check

A software check is used to check the existence of software on the target environment. For example, we can check for the existence of IBM DB2 Universal Database Version 7.2 or later. To specify a software package, it needs to be referenced by a UUID or name, or both. Minimum and maximum versions can be specified.

Example 3-15 Software check element

```
<software checkVarName="db2_for_Linux_check">
   <UUID>12345678901234567890123456789012</UUID>
   <name pattern="true">('DB2' or 'Universal Database')</name>
   <minVersion>7.1</minVersion>
</software>
<software checkVarName="JRE_for_Windows_check">
   <UUID>12345678901234567890123456 7</UUID>
   <name pattern="true">(IBM+.*J2RE)</name>
   <minVersion>1.3.1</minVersion>
   <maxVersion>1.4.1</maxVersion>
</software>
```

Custom check

A custom check allows you to run a custom command that is specified in the check. When you define a custom check you need to define the following element in the descriptor:

- customCheckDefinition - Defines the command to run for the check, and defines the meaning of the return codes

Example 3-16 Custom check element

```
<custom customCheckIdRef="WindowsRegistryCustomCheck"
   checkVarName="GSK4_Win_Registry_Check">
   <parameter variableNameRef="Win_Reg_Hive">HKEY_LOCAL_MACHINE</parameter>
   <parameter variableNameRef="Win_Reg_Key">
   SOFTWARE\IBM\GSK4\CurrentVersion\Version
   </parameter>
   <parameter variableNameRef="Win_Reg_Type">REG_SZ</parameter>
   <parameter variableNameRef="Win_Reg_Value">4.0.2.49</parameter>
</custom>
```

The above example assumes WindowsRegistryCustomCheck to be the identifier of an external check command defined in the software package. This command may be implemented to access the Windows registry and check the value of a generic registry key for equality against a specified value. As for any other check, the variable associated with the check result is defined by the checkVarName attribute (GSK4_Win_Registry_Check). The parameter elements are used to assign a value to the four arguments that must be passed to the external check command.

Constraints

The constraints is an *optional* element of the IU definition. The following two elements can be specified:

- maximumInstances - This element defines the limit of the number of instances of the IU that can be deployed onto the same hosting environment. No limit exists when the element is not specified.
- maximumSharing - This element limits the number of dependent IU instances that can have a "uses" relationship with this IU.

Both the elements are non-negative integers.

Example 3-17 Constraints element

```
<constraints>
   <maximumInstances>1</maximumInstances>
   <maximumSharing
   sharedBy_List="db6c9c23b63f5af0f26dc819cd9a2786">100</maximumSharing>
</constraints>
```

The above example imposes the constraint that there must be only one instance of the IU being defined within its hosting environment. It also dictates that there could be a maximum of 100 dependent instances sharing the IU. All of them must be instances of IUs with the specified UUID.

Consolidated fixes

Note that this capability is not supported in the current release.

One SIU may be an upgrade to a previous version of the same component, in which case it is identified by the same UUID of the IU being upgraded plus a different version.

Let us assume that an SIU Version 1.2 can be installed as an upgrade of both Version 1.0 and Version 1.1 as well as a fresh install of Version 1.2.

An IU will be installed as an upgrade of a deployed instance if the instanceIdentifier (for a unit targeted to the OS hosting environment this is the install location) is equal to the instance identifier of a previously installed instance of the same component (same UUID) whose version is declared among the unit's software requirements.

Temporary fixes may have been applied to the IU before the upgrade. It is important to know the level of fixes that one can assume to be consolidated by the IU upgrade. This information is provided by the element consolidatedFixes of the IU definition. Each element in the list is a reference to a temporary fix based on the fixName attribute of the latter.

Example 3-18 ConsolidatedFixes element

```
<consolidatedFixes>PTF012345 PTF123456 PTF234567</consolidatedFixes>
```

3.1.6 Variables element

As you might guess, this element contains information about variables. Variables include parameters, which are optionally initialized with a default value and possibly overridden by user input. Other variables take their value from a variable expression or from a query of a hosting environment property. Several elements appearing within an SIU are instances of the base:VariableExpression type.

Any variable that is referenced by a variable expression within an SIU must be declared within the SIU or any of its parent aggregating IUs. The variable must simply be in scope to be used in the SIU. However, a variable may be assigned a value by user-input (override of a parameter default).

One or more variables may be defined by the variables element within an installable unit. This is a sequence of one or more variable elements. Each one identifies a named variable that is required to deploy the installable unit. The variable may have a locale-sensitive description. The value of the variable is obtained by one of the following:

- User input, response file, or from an encapsulating solution module.
- Derived from a variable expression.
- Query against a target.
- Query against an IU instance, returning the IU instance discriminant. This is the value that uniquely identifies an IU instance within a hosting environment, for example, the install location for an IU installed into the operating system.
- Specified as a resolved target list.

The only attribute that a variable element has is *name*. This is a required attribute and identifies the symbolic variable name.

Example 3-19 Declaring a variable as parameter

```
<variables>
   <variable name="IsInstallDir">
      <parameter defaultValue="true"/>
   </variable>
</variables>
```

Derived variable

A derived variable is defined through a sequence of expression elements. A derived variable has two optional attributes, *condition* and *priority*. The attribute "condition" is not needed when there is only one expression. However, when multiple expressions are present, each expression must specify the "condition" attribute, which is interpreted as a conditional expression.

Attribute “priority” is a number determining the expression value to be assigned to the variable when multiple expression elements are defined and their conditions are not mutually exclusive.

Example 3-20 Derived variable

```
<variable name="install_root">
   <derivedVariable>
         <expression condition="$(Windows_Check)">C:\Program
Files</expression>
   <expression condition="$(Linux_Check)">/usr/opt</expression>
   </derivedVariable>
</variable>
```

In the above example, the variable named “install_root” is set to the value “C:\ProgramFiles” or to “/usr/opt” depending on alternative conditions. In this example, it is assumed that the variable expression associated with each condition is the simple value (“true” or “false”) of an elementary check. It is also assumed that the two checks cannot be simultaneously satisfied on the same hosting environment; therefore there is no need to specify a priority.

Query target

This variable returns the value as a result of a property query.

Example 3-21 Query target variable

```
<variable name="VirtMemSize">
   <queryTarget property="totalVirtualMemorySize" />
```

```
</variable>
```

In the above example, the variable named “VirtMemSize” is set to the value of the total virtual memory size property of the operating system hosting environment.

Resolved target list

A variable defined by this element will be assigned a value containing the list of physical targets corresponding to a logical target.

Example 3-22 Resolved target list

```
<variable name="Actual_Targets">
   <resolvedTargetList target="Solution_Server" />
</variable>
```

3.1.7 Unit element

The unit element is unique to a smallest installable unit. It defines the artifacts that can be targeted at the hosting environment to manage the life-cycle of the installable unit. Each unit element has a condition attribute that determines whether the artifact set is applicable to an instance of a logical target. The life-cycle operations (such as create, configure, delete, undo, and so forth) are implemented through actions. The actions to be performed for each operation are typically defined in a separate file known as an action artifact.

The unit element in a smallest installable unit defines the artifacts to be used when performing change management operations on the installable unit. Details of action artifacts are given in 3.3.1, “Action artifacts” on page 65.

The unit element has an attribute *condition*. This attribute is an optional variable expression and is a way of defining a conditional expression. After variable substitution has occurred within a condition, the resulting conditional expression must be a constant expression.

When there is only one expression, this attribute does not need to be defined. When multiple expressions are present, however, each expression must specify the condition attribute. The condition attribute is interpreted as a conditional expression.

Below is the fragment of deployment descriptor having a unit element. The fileIdRef element refers to the actual artifact file, which is declared in a *file* element. See 3.1.10, “File and files elements” on page 56, for details.

Example 3-23 Unit element

```
<unit>
   <installArtifacts>
      <installArtifact>
         <fileIdRef>HelloWorldInstallActions</fileIdRef>
         <parameterMaps>
            <map>
               <internalName>InstallDir</internalName>
               <externalName>InstallDir</externalName>
            </map>
         </parameterMaps>
      </installArtifact>
      <uninstallArtifact>
         <fileIdRef>HelloWorldUninstallActions</fileIdRef>
         <parameterMaps>
            <map>
               <internalName>InstallDir</internalName>
               <externalName>InstallDir</externalName>
            </map>
         </parameterMaps>
      </uninstallArtifact>
   </installArtifacts>
</unit>
```

3.1.8 Topology element

The topology element defines a set of target definitions and a set of deployed targets. The targets that must exist are the ones required by the selected installable units (that is, the target for the installable unit or those identified by a target reference in a check within the installable unit). The element associated with this activity is the topology element.

Target element

The solution module definition identifies a set of target manageable resources that defines the target topology for solution module deployment. Targets may be referenced within an IUDD to identify:

- An installable unit's hosting environment
- The target of a property query
- The target of a check

The target definition has the following attributes:

- An identifier that is used to refer to this target from within the same installable unit package. Targets are not referenced from independently packaged

installable units. To correlate between targets in different packages, a target map is used.

- The type category for this target. This represents the logical resource type, such as a J2EE application server. Standard values are predefined for this in resourceTypes.xsd.

The target definition can contain several other elements, some of which include:

- The target scope, which identifies how to resolve the target:
  - One: Resolve to a specific instance that satisfies target dependencies and relationships. This is also the default value for target scope.
  - All: Resolve to every matching instance. (Not currently supported.)
  - Some: Resolve to a subset of matching instances. (Not currently supported.)
- An optional member element, which may be used to set the initial selection of targets from the union of two or more other targets. If any member elements are present, then the selection dependencies are applied only to the union of the referenced target sets.
- The set of selection dependencies that are used to select the targets in this target set.
- The set of validation dependencies that are used to validate targets that have been selected in this target set. These dependencies are used to verify that each target meets certain criteria; they also provide information that is needed to modify the target so that it meets these requirements. The validation dependencies are also used to assist in further selection from the target set if the scope is one or some.

Note: If scope is one or some and there is more than one match, the deployment application may apply some pre-defined policy, select from matching targets based on validation dependencies, or interact with the user to make the selection.

Example 3-24 Topology element extract

```
<topology>
      <target id="tOperatingSystem" type="OSRT:Operating_System">
         .
         .
         .
      </target>
</topology>
```

SelectionRequirements element

The target element can also contain selectionRequirements elements, which are a collection of *requirement* elements. We have already seen the requirement element and its contents like alternatives and checks.

Example 3-25 Requirement element

```
<topology>
   <target id="tOperatingSystem" type="OSRT:Operating_System">
      <selectionRequirements>
            <requirement name="OS">
               <alternative name="Windows_2000_Check">
                  <inlineCheck>
                     <property checkId="Windows_2000_Check">
                        <propertyName>OSType</propertyName>
                        <value>Windows 2000</value>
                     </property>
                  </inlineCheck>
               </alternative>
               <alternative name="Windows_XP_Check">
                  <inlineCheck>
                     <property checkId="Windows_XP_Check">
                        <propertyName>OSType</propertyName>
                        <value>Windows XP</value>
                     </property>
                  </inlineCheck>
               </alternative>
               <alternative name="Linux_Check">
                  <inlineCheck>
                     <property checkId="Linux_Check">
                        <propertyName>OSType</propertyName>
                        <pattern>LINUX.*</pattern>
                     </property>
                  </inlineCheck>
               </alternative>
            </requirement>
         </selectionRequirements>
   </target>
</topology>
```

Requirements, alternatives, and check elements can be placed inside a topology element to select an appropriate hosting environment for the installation.

3.1.9 RootInfo element

The rootInfo element contains information about the deployment descriptor and the schema against which the descriptor was built.

A rootInfo element must provide the following details:

- Schema version of the schema files used to create the valid deployment descriptor.
- A build element is contained within the rootInfo element and indicates the version of the software package itself separate from the versions of the installable units that the software package contains.
- Size of the installable unit in kilobytes.
- The bigIcon element is an optional element that is contained within the rootInfo element and indicates the referenced GIF file of the 32x32 icon associated with the software package.

Example 3-26 rootInfo element

```
<rootInfo>
   <schemaVersion>1.2.0</schemaVersion>
   <build>42</build>
   <size>1457</size>
</rootInfo>
```

3.1.10 File and files elements

The rootIU element contains the *files* element. The *files* element contains one or more file elements that contain the definition for each file in the package. Where files do not need to be individually referenced in the descriptor, they may be distributed in an archive format, with a single entry in the descriptor.

A file element has the following attributes:

- The ID value is used as a key to reference the file element from other types.
- An optional compression boolean attribute specifying whether the file needs to be automatically compressed during packaging and automatically decompressed before being used by actions (default is “true”).
- An optional charEncoding.
- An optional language.

A files element has the following elements:

- The path name of the file image within the package volume or folder.
- The length of the file image as stored within the package (if the file “compression” attribute is specified, this is the length of the compressed image).
- A checksum to be used for validation.

Example 3-27 Files and file elements

```
<files>
      <file id="HelloWorldInstallActions">
         <pathname>HelloWorldInstallArtifacts.xml</pathname>
         <length>1431</length>
         <checksum type="MD5">26e91abc6d82a94f96af05316ba8ee01</checksum>
      </file>
      <file id="HelloWorldFile1" compression="false" charEncoding="US-ASCII">
         <pathname>../FILES/HelloWorld.txt</pathname>
         <length>14</length>
         <checksum type="MD5">6d82a94f96af05316ba8ee0126e91abc</checksum>
      </file>
      <file id="HelloWorldUninstallActions">
         <pathname>HelloWorldUninstallArtifacts.xml</pathname>
         <length>1247</length>
         <checksum type="MD5">f05316ba8ee0126e91abc6d82a94f96a</checksum>
      </file>
</files>
```

3.1.11 Feature element

Features provide the mechanism that enables a user to select which installable units in the root installable unit to deploy. Do not confuse features with conditions. Conditions filter which installable units in the root installable unit to deploy based on environmental properties.

Features are defined by a feature element. This element requires a unique value for the name attribute and optionally takes a value for the displayName attribute. The value for the name attribute must be unique within the deployment descriptor for the root installable unit. The value for the displayName attribute is a descriptive name for the feature.

A feature element can contain one or more of the following definitions:

- Nested sub features.
- Installable units that are selected by an IUNameRef from a top-level installable unit. These top-level installable units must be defined within the selectableContent of the root installable unit.
- Reference to features defined within referenced installable units.
- Optional selection rules. Each selection rule identifies the name of a different feature within the root installable unit. The specified featured is the subject of the selection rule. The following selection rules are available:
  - Select when selected
  - Select when deselected

– Deselected when deselected
– Deselected when selected

Restriction: Note that selection rules are not supported in Version 1.2 of the Solution Installation toolkit.

Example 3-28 Feature element example

```
<features>
      <feature featureID="BFeature">
         <identity>
            <name>Feature B</name>
         </identity>
         <IUNameRef>MyWorld</IUNameRef>
         <selectionRules>
            <deselectIfSelected>BFeaturePlus</deselectIfSelected>
         </selectionRules>
      </feature>
      <feature featureID="BFeaturePlus">
         <identity>
            <name>Feature B Plus</name>
         </identity>
         <IUNameRef>MyWorld</IUNameRef>
         <IUNameRef>HelloWorld</IUNameRef>
         <selectionRules>
            <deselectIfSelected>BFeature</deselectIfSelected>
         </selectionRules>
      </feature>
</features>
```

Features are associated with installable units. For each defined feature, the defined installable units can be partitioned into required content and selectable content. Required content indicates which installable units to deploy independent of what the user selects. Selectable content indicates which installable units to deploy based on what a user selects.

3.1.12 Installation groups

Installation groups are used to define a set of features that should all be installed if the group is selected. A group consists of:

- The name of the group.
- A list of the features that should be selected for installation if the group is selected. These names must be IUNames defined within the package.
- A list of the mapped features within packaged installable units that should be selected for installation if the group is selected.

- An optional human-readable description.

All features within a member of the installation group are selected when that group is selected. A feature can be a member of multiple groups, or of no groups. A group cannot contain other groups.

Another use of installation groups is to provide installation based on roles (developer, administrator, user) or on other usage criteria (typical, compact, custom). In the latter case, installation groups would be defined for the typical and compact installations, and the custom installation would allow the user to select from, for example, a set of features.

An example of using installation groups is shown below.

Example 3-29 Installation groups

```
<groups>
   <group>
      <groupName>Group1</groupName>
      <feature featureIDRef="Feature_Code_A" selection="selected" />
      <feature featureIDRef="Feature_Code_B" selection="selected" />
      <feature featureIDRef="Feature_Documentation_B" selection="selected" />
   </group>
   <group>
      <groupName>Group2</groupName>
      <feature featureIDRef="Feature_Code_A" selection="selected" />
      <feature featureIDRef="Feature_Code_B" selection="selected" />
   </group>
   <group>
      <groupName>Group3</groupName>
      <feature featureIDRef="Feature_Documentation_B" selection="not_selected"
/>
   </group>
   <default>Group1</default>
</groups>
```

This schema does not require installation groups to be used, nor does it define how the user should select from the groups (for example, which are required or optional). The appropriate interpretation is left to the interactive installation program.

Features and installation groups determine which installable units to deploy, but they do not determine the order in which the installable units should be deployed. The deployment sequence is determined by the following characteristics:

- The hierarchy of the installable units.

- The sequence number associated with the installable unit. (Not supported in the current release.)

During the installation process, features are registered, while installation groups are not registered. Because features are registered, you can define dependency checks against the features. Each feature has a unique name within a root installable unit.

3.1.13 Updates and fixes

Installable units can be upgraded as a full update, an incremental update, or a fix. Updates and fixes can occur at any level of aggregation, including a root installable unit. Updates and fixes follow a hierarchy that must match the hierarchy or the original root installable unit. These hierarchies must match so that a consistent set of rules can be applied to both updates and fixes.

At each level of the hierarchy, there can be multiple updates and fixes that correspond to each of the original installable units in the deployment descriptor. Updating an in line (or embedded) installable unit means that all of its parents, up to its root installable unit, are considered to be updated.

A fix is very similar in content to that of an SIU or CIU, with the following differences:

- A fix is identified by a fix name. It does not have a UUID.
- The fix identifies the UUID and versions of the installable unit that it fixes.
- A fix is uniquely identified by the UUID of the installable unit that it fixes, and its fix name.

The following characteristics apply to the support for updates and fixes:

- Requirements can be placed on a fix at the solution module or container installable unit level.
- Updates and fixes are applied in a given context within their aggregating installable unit.
- A set of fixes against a solution can only be installed when the solution was installed together. As a corollary, in line installable units within a solution cannot be updated by fixes for the same installable unit in another solution.

3.2 Media descriptor

The deployment descriptor specifies what files are part of the package, but it does not define where those files reside. The media descriptor provides the

binding information about the physical location of these files. Separating the deployment descriptor from the media descriptor allows the same deployment descriptor to be used for different physical packages. Each physical package can have a media descriptor that provides binding information that describes specifically where each file is physically located.

For instance, an installation of an application could have a standard deployment descriptor but different media descriptors if the installation files are on CD, on a network file server, or available via the Internet or some other media.

The media descriptor is optional. If it is not present, all files specified in the deployment descriptor are assumed to have paths that are relative to the location of the deployment descriptor in the package. A deployment engineer might choose to override this default value using, for example, an installation parameter.

There is, at most, one media descriptor per package. When the package is aggregated into another package, the referenced deployment descriptor can have its own media descriptor.

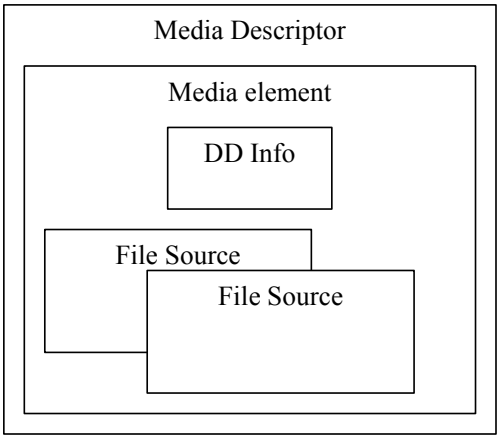

Figure 3-5 Block diagram of a media descriptor

The media descriptor defines the following information:

- The path name of the deployment descriptor relative to the media descriptor.
- A default logical source that can be applied to files that are not explicitly bound in this media descriptor.
- The physical location of the files in the package.
- Any overrides to file paths for specific files. This function can be used to map common files that are shared by multiple installable units.

At the beginning of this chapter we have seen the schema files that are used to define a deployment descriptor. There is another schema file, media.xsd, which defines the syntax and grammar of the media descriptor.

3.2.1 Media descriptor information

Like the rootIU element in deployment descriptors, the media descriptor has *binding* as its root element. This element contains all the binding information about the schema. The sub elements of media are:

- Deployment descriptor information
- File source

The deployment descriptor element

A media descriptor has the following deployment descriptor information:

- The deployment descriptor physical path
- The optional message digest for the entire deployment descriptor

The physical path is relative to the media descriptor and links the media descriptor to the associated deployment descriptor. The optional digest value is used for security purposes.

Example 3-30 Deployment descriptor information in a media descriptor

```
<deploymentDescriptor>
    <location>packagedSM.xml</location>
    <digest type="MD5">f96af05316ba8ee0126e91abc6d82a94</digest>
</deploymentDescriptor>
```

File source element

A media descriptor contains the following information regarding the files that make up the installable unit:

- fileIdRef - This is same as the file ID in the file element of a deployment descriptor.
- location - Location of the file.

Files specified in a deployment descriptor have paths relative to logical sources. A logical source, depending on the package types, can be any physical location. This allows the files to be packaged in a way that meets user scenarios and needs.

The default value of a logical source is the location of the deployment descriptor. If the media descriptor is not present or the installer does not specify the physical location of a logical source, the default value is used.

A media descriptor can specify the physical location of the logical source, and can optionally override the file paths. The physical location of a file's logical source can be one of the following:

- Local path: This is a path relative to the media descriptor folder (which could be in a ZIP file, file system, network location, or CD).
- A folder in a ZIP file: ZIP is a commonly used compression format, but the access to the contents of a ZIP file is different from the access to a regular file system—the process requires decompression. Locations of this type need to be specified so installers can properly access the files.
- Location in a fixed-sized removable media such as a CD-ROM: Such a location is identified through the volume identifier and a path in the volume.
- Network location: This is any URI that allows the file to be anywhere on the network. This can be used to locate a deployment descriptor of an external aggregated IU package that exists on the network.

Each file entry in the deployment descriptor can have a corresponding <fileSource> entry in the media descriptor. The file IDs in the deployment descriptor are used to identify the entries.

Example 3-31 FileSource element for an individual file

```
<fileSource>
    <fileIdRef>file_245</fileIdRef>
    <location>
      <physicalLocation>
        <!-- Relative to the media descriptor folder (META-INF/ in the ZIP
        file). -->
        <!-- Suppose the path for 'file_245' is dir/foo.java, the path in
          will be ../FILES/dir/foo.java -->
        <local>../FILES</local>
      </physicalLocation>
    </location>
</fileSource>
```

A media descriptor can only specify the physical locations for the logical sources, or override the file path. It cannot modify the file sizes and digest values.

Note: Not all of the files are bound in the media descriptor. Some files might be bound later either via a modified media descriptor or directly by installation parameters. Otherwise, the default values will be used. For those scenarios, the physical locations are simply unknown. A file path can be overridden in the media descriptor. If the file path is specified in the media descriptor, the one in the deployment descriptor will be ignored.

Example 3-32 FileSource element for a zipped file

```
<fileSource>
   <!-- Reference to a 'file' entry in the deployment descriptor -->
   <fileIdRef>file_776</fileIdRef>
   <location>
      <physicalLocation>
         <ZipFile>
            <removableMedia type="CDROM" volumeNumber="2"/>
            <!-- The folder (relative to the root) inside the Zip file. -->
            <!-- Suppose the path for 'file_776' is lib/audio.so (defined in
            the deployment descriptor) and the network location for the
            descriptors are at http://depot.my.com/applicationA, then the path
            for 'file_776' is in http://depot.my.com/pkg/Library.zip as
            /FILES/lib/video.so -->
            <path>FILES</path>
         </ZipFile>
      </physicalLocation>
   </location>
</fileSource>
```

Example 3-33 FileSource element for a file in network location

```
<fileSource>
   <!-- Reference to a 'file' entry in the deployment descriptor -->
   <fileIdRef>file_67</fileIdRef>
   <location>
      <physicalLocation>
      <!-- A network location URI. -->
      <!-- Suppose the path for 'file_67' is lib/sound.so (defined in
      the deployment descriptor) and the network location for the
      descriptors are at http://depot.my.com/applicationA, the location
      for 'file_67' is
      http://depot.my.com/applicationA/FILES/lib/sound.so. -->
         <URL>FILES</URL>
      </physicalLocation>
   </location>
</fileSource>
```

Example 3-34 FileSource element for a file in removable media

```
<fileSource>
   <!-- Reference to a 'file' entry in the deployment descriptor -->
   <fileIdRef>file_776</fileIdRef>
   <!-- Indicate that the file is located inside an external ZIP file on a
   CD. -->
   <location>
      <physicalLocation>
         <ZipFile>
```

```
            <removableMedia type="CDROM" volumeNumber="2"/>
            <!-- The folder (relative to the root) inside the Zip file. -->
            <!-- Suppose the path for 'file_776' is lib/audio.so (defined
               in the deployment descriptor) and the network location for
               the descriptors are at http://depot.my.com/applicationA,
               then the path for 'file_776' is in
               http://depot.my.com/pkg/Library.zip as
               FILES/lib/video.so-->
            <path>FILES</path>
         </ZipFile>
      </physicalLocation>
   </location>
</fileSource>
```

Example 3-35 FileSource element for files whose location is unknown

```
<fileSource>
   <!-- Reference to a 'file' entry in the deployment descriptor -->
   <fileIdRef>file_777</fileIdRef>
   <!-- Indicate that the file physical location is not known -->
   <location>
      <physicalLocation>
         <unknown/>
      </physicalLocation>
   </location>
</fileSource>
```

3.3 Artifacts

There are two types of artifacts that are used in the Solution Installation framework.

- Action artifacts: To define the actions to be carried out during the various change management operations
- Configuration artifacts: Used for hosting environment configuration during post install/uninstall

3.3.1 Action artifacts

We have already seen that the *unit* element in a smallest installable unit can define the action artifact files. This section describes the contents of an action artifact file.

It should be noted that actions in the artifact are specific to a type of hosting environment, and not part of the base IUDD definition. Third parties can define new actions without changing the IUDD.

An action artifact is a descriptor containing the definition of the actions to be performed for various change management operations. The separation of the action definition from the declarative installable unit dependency information allows a supporting implementation where the deployment descriptor is interpreted by a generic change management component and the actions are interpreted by a hosting environment-specific component, the touchpoint. For more details on operations and actions refer to Chapter 4, “Solution Installation operations” on page 77.

The action artifact file is specified in the unit element in the deployment descriptor. The action artifact files also need to be mentioned in a file element.

Example 3-36 Action artifact file declared in a file element

```
<file id="HelloWorldInstallActions">
          <pathname>HelloWorldInstallArtifacts.xml</pathname>
          <length>1431</length>
          <checksum type="MD5">f96af05316ba8ee0126e91abc6d82a94</checksum>
</file>
<file id="HelloWorldUninstallActions">
          <pathname>HelloWorldUninstallArtifacts.xml</pathname>
          <length>1247</length>
          <checksum type="MD5">abc6d82a94f96af05316ba8ee0126e91</checksum>
</file>
```

The following figure represents the elements in the action artifact files.

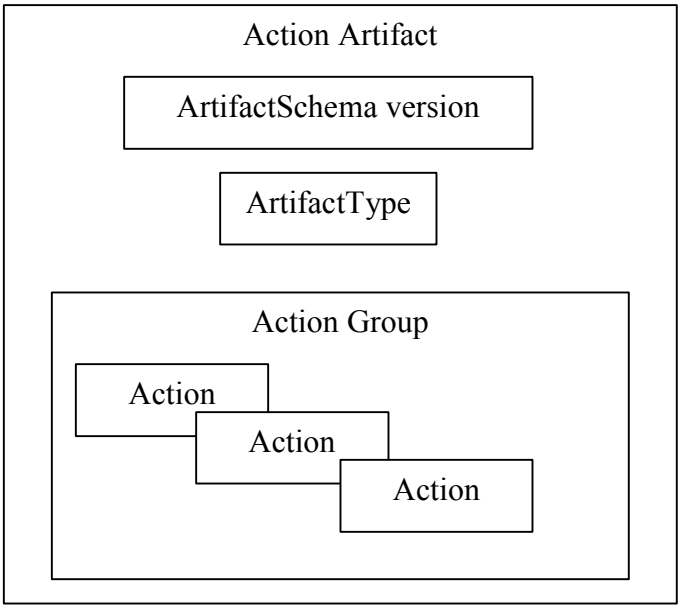

Figure 3-6 Block diagram of the elements in an action artifact file

<action:artifact> is the root element of this XML file. It contains information about the schema name spaces and location. The sub elements of the root element are artifactType, artifactSchemaVersion and actionGroup.

The artifactType element can take one of several values. They loosely correspond to the change management operations; however, there can be some confusion because artifactType names are similar but not identical to the change management operation names.

The artifactType names include:

- Install
- InitialConfig
- Migrate
- VerifyInstall
- Uninstall

Action artifacts are applied to a hosting environment to implement a software life-cycle operation.

The following is extracted from the top of an action artifact file.

Example 3-37 Extract from an action artifact

```
<action:artifact
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oca="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/act
ion action.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions osActions.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE base.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/command command.xsd">
   <artifactType>Install</artifactType>
   <artifactSchemaVersion>1.2.0</artifactSchemaVersion>
   .
   .
   .
```

ActionGroup element

The actionGroup element can contain many *actions* elements that in turn can contain various pre-defined actions for the target touchpoint. In the example below, the actions are defined for an OS touchpoint.

Example 3-38 ActionGroup element and its sub elements

```
<actionGroup xsi:type="osac:OsActionGroup">
   <actions>
      <addDirectory>
         <directory>
            <location>./</location>
            <name>HellowWorldDir</name>
         </directory>
      </addDirectory>
      <addFile>
         <file>
            <location>./HelloWorldDir</location>
            <name>HelloWorld.txt</name>
            <source>
               <fileIdRef>HelloWorldFile1</fileIdRef>
            </source>
         </file>
      </addFile>
   </actions>
</actionGroup>
```

The following basic actions are supported by the operating system hosting environment.

- addFile
- removeFile
- addDirectory
- removeDirectory
- addLink
- removeLink
- executeExternalCommand
- executeJavaClass

Actions creating files and directories have attributes associated with them, including the following:

- replace_if_existing: Specifies to replace an object that already exists on the target. This is a boolean attribute; use is optional attribute and default is false.
- replace_if_newer: Specifies to replace a target object (file or directory only) even if the target object is newer than the source object. This is a boolean attribute; use is optional and default is false.
- remove_if_modified: Specifies to remove an object even if the target object has been modified. This is a boolean attribute; use is optional and default is false.
- remove_empty_dirs: Specifies to remove the directory containing a file that was removed. This is a boolean attribute; use is optional and default is true.
- remove_extraneous: Specifies whether to remove all files and directories in an existing destination directory prior to installing or removing a unit of software. This is a boolean attribute; use is optional and default is false.
- create_dirs: This can be used with Directory and File actions to specify that directories should be created if they do not already exist on the target system. This is a boolean attribute; use is optional and default is true.
- delta_compressible: Specifies whether files must be delta compressed or not. This attribute can be specified at the file as well as at the directory level. If set to “yes”, it indicates that the file must be delta compressed. If set to no, it indicates that the file must not be delta compressed. If set to default, indicates that the file must be delta compressed only if it satisfies some rules.
- temporary: It can be used with Directory and File actions to specify whether the object must be removed during cleanup. This is a boolean attribute; use is optional and default is false.

Note: It should be noted that file and directory actions can become relatively complex to manage. For instance, when multiple installable units require (or create) the same directory or file, the deletion of a directory or file when one installable unit is uninstalled may affect the operation of other installed packages.

It is possible to deal with such situations through use counts and other techniques, but such a discussion is beyond the scope of this book.

Two key actions that should be highlighted are the executeJavaClass and executeExternalCommand. These commands can be used to launch an existing installation program on the target system. That is, for software packages that exist today with their own installation program that can be run in unattended mode, the action artifact to install that product may only need to use the executeExternalCommand, for instance, to install the package. Therefore, it is not necessary to recreate an existing installation process through complex sequences of touchpoint actions.

AddDirectory element

The add directory element, <addDirectory>, adds a directory in the hosting environment. The directory element is defined within the addDirectory element. It has a location element that defines the parent directory under which this directory is to be created.

Within the directory description, the attributes for the directory element can be inserted to indicate how to process this action based on specific data, such as modification date or existence. When describing the directory, the following elements are used:

- location - This element specifies the directory under which a new directory will be created.
- name - This element indicates the name of the new directory. This value can include additional subdirectories to append to the value indicated by the location element.
- source - This element indicates whether the source is a directory or an archive. For a directory, it indicates a directory to copy to the target system. For an archive, it indicates an archive (in zip or jar format) to extract to the target system.

Example 3-39 AddDirectory element example

```
<addDirectory>
   <directory>
      <location>./</location>
```

```
      <name>helloworld</name>
   </directory>
</addDirectory>
```

AddFile element

When added to the action artifact, the addFile element causes a file to be added to the hosting environment. Within the addFile element, the file element is used.

The file element has the following attributes.

- transcoding - This is a boolean attribute. When “true”, this attribute specifies that the file to be created is a text file, and a character encoding change must be performed on the source file to create the target file. The source file can be a packaged file, in which case the character encoding can be specified by the charEncoding attribute of the packaged file. In all other cases, it is assumed and required that the source file be encoded in UTF-8. When “true”, this attribute requires the attribute charEncoding to be specified.

  Note: Transcoding is not supported in Version 1.2 of the toolkit.

- charEncoding - This attribute identifies a file as containing text. It specifies the character encoding of the file by the IANA character set name. When used in conjunction with the transcoding attribute, it specifies the file’s final character encoding.
- language - This attribute is an instance of the XML base “language” type. It is used to specify that a file contains text in one specific natural language.
- source - This element specifies the source of the file, and is a choice among the following two elements:
  - fileIdRef - This element is specified for an archive file that is packaged together with the software unit. This element is a reference to a file defined within the package.
  - path - This element is specified for an archive that must be found on the file system of the target at the specified absolute path.

Example 3-40 AddFile element

```
<addFile>
   <file>
      <location>./helloworld</location>
      <name>HelloWorld.txt</name>
      <source>
         <fileIdRef>HelloWorldFile1</fileIdRef>
      </source>
   </file>
```

```
</addFile>
```

AddLink element

When added to the action artifact, the addLink element causes two files to be linked. There are two attributes that are required in addition to the common ones discussed above.

- source_file - This element specifies the full path name of the link source file.
- destination - This element specifies the full path name of the link.

RemoveDirectory element

The removeDirectory element, when defined in a deployment descriptor, causes a directory to be removed. When the removeDirectory element is defined, the name element must be specified. And optionally the displayName element and the location element can be specified.

Example 3-41 RemoveDirectory element

```
<removeDirectory actionSetIdRef="osIBMBase">
      <location>./</location>
      <name>helloworld</name>
</removeDirectory>
```

RemoveFile element

The removeFile element, when defined in a deployment descriptor, causes a file to be removed. When the removeFile element is defined, the name element must be specified. And optionally the location element can be specified.

Example 3-42 removeFile element

```
<removeFile>
      <location>./helloworld</location>
      <name>HelloWorld.txt</name>
</removeFile>
```

RemoveLink element

The removeLink, when defined in a deployment descriptor, causes a link to be removed. When the removeLink element is defined the name element must be specified. It has an attribute pathname. This element specifies the full path name of the link.

Note: Even if the link action is specified within an enclosing addDirectory element, the link pathname element does not represent a relative path with respect to the parent directory.

External command action

In addition to those basic actions listed above, Solution Installation also provides a way to run external commands. Each one of the additional elements of an external command defines the command to be executed for a specific change management operation or an internal—backup or cleanup—operation.

The backup element specifies the external command to be executed during the backup operation. The cleanup element specifies the external command to be executed during cleanup operation after install/uninstall.

Example of uninstall action artifact

Below is an example of a simple (but complete) action artifact for an uninstall operation.

Example 3-43 Uninstall artifact example

```
<?xml version="1.0" encoding="UTF-8"?>
<action:artifact
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oca="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/act
ion action.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions osActions.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE base.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/command command.xsd">
   <artifactType>Uninstall</artifactType>
   <artifactSchemaVersion>1.2.0</artifactSchemaVersion>
   <actionGroup xsi:type="oca:OsActionGroup">
      <actions>
         <removeFile>
            <location>./helloworld</location>
            <name>HelloWorld.txt</name>
         </removeFile>
         <removeDirectory>
            <location>./</location>
            <name>helloworld</name> n
         </removeDirectory>
      </actions>
   </actionGroup>
</action:artifact>
```

3.3.2 Configuration artifacts

Configuration artifacts are used to alter the configuration of an installed package. In addition to the configuration-specific elements described below, it should also be noted that a common way of performing configuration is to use an action artifact that launches a configuration script.

Configuration unit

A configuration unit is an atomic unit of configuration that can be specified as part of an aggregated installable unit. It differs from an installable unit in the following ways:

- A configuration unit does not have a unique identity and is not registered.
- A configuration unit contains artifacts that can be applied to any resource, not just the hosting environment.
- The artifacts in a configuration unit are applied post-installation using the configuration and verify configuration change management operations.

Like other installation units, a configuration unit can have checks, requirements, and variables. It can be interleaved with other installation units within a rootIU. A configuration unit has the following features:

- It is applied post-install by a ‘configure’ change management operation. Change management operations are discussed in Chapter 4, “Solution Installation operations” on page 77.
- It may be reapplied multiple times.
- It can be verified with the VerifyConfig change management operation.
- Like other installable units, it can have artifacts associated with change management operations.
- It may have multiple, mutually exclusive artifacts per change management operation based on conditions.
- It can target any resource not just the hosting environment.

ConfigurationUnit element

The configurationUnit element defines the configuration that can be included in an aggregated installable unit. When defined in a deployment descriptor, it contains sub elements such as the CUDefinition element, the variables element, and the unit element.

The configuration unit contains the following elements:

- CUDefinition - The CUDefinition element defines the configuration unit. When defined in a deployment descriptor, specify the following elements, as needed:
  - displayName - The displayName element is an optional element that includes a sequence of definitions, each one in the specified language, for the display name and description of the installable unit.
  - manufacturer - The manufacturer element is an optional element that includes a sequence of definitions, each one in the specified language, for the manufacturer of the installable unit.
  - checks - The checks element is an optional element that includes a sequence of checks to perform.
  - requirements - The requirements element is an optional element that includes a sequence of requirements to evaluate.
- variables - The variables element contains information about the variables.
- unit - The unit element is a required element that contains the artifacts that comprise the configuration unit. The configuration unit can contain two types of configuration artifacts:
  - The configuration artifact
  - The verify configuration artifact

  The configuration artifact (the configArtifact element) contains the definition of the configuration to be applied. The verify configuration artifact (the verifyConfigArtifact element) contains the definition of the configuration to be verified. The schema definition of these files can be found in the config.xsd file.

The CUDefinition element has the following attributes.

- CUName - The CUName attribute is a required attribute that is an identifier of the configuration unit. This attribute is not a persistent global identifier. This attribute can be used to refer to the configuration unit from other elements within the deployment descriptor.
- resourceType - The resourceType attribute is an optional attribute that specifies the type of resource to which the configuration unit is to be applied. A configuration unit contains a configuration definition that can be specific to the target resource.

Configuration artifact

A configuration artifact, like an action artifact, is based on the osActions.xsd schema file. The configuration artifact handles the operation to configure an

installable unit after installation. These artifacts perform the following types of operations:

- Reconfiguration
- Verification of the configuration

A configuration artifact specifies values for properties of the target resource.

Example 3-44 Sample configuration artifact

```
<?xml version="1.0" encoding="UTF-8"?>
<config:configArtifact
xmlns:config="http://www.ibm.com/namespaces/autonomic/solutioninstall/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/con
fig config.xsd ">
  <propertyValues>
    <NumberOfLicensedUsers>5</NumberOfLicensedUsers>
  </propertyValues>
</config:configArtifact>
```

3.4 Summary

This chapter has introduced deployment descriptors and other artifact files. These are XML files whose schemas provide a standard way to declare information about installable units. The runtime components of the Solution Installation framework consume these files to verify pre- and co-requisites, perform a variety of checks to ensure the installation should continue, and to perform the appropriate actions for a specific change management operation.

In the following chapter, we introduce the basic change management operations that are supported by Solution Installation.

4

Solution Installation operations

In the last chapter we covered various descriptor files that describe installable units and define how a change manager should handle a change management request related to the IU.

The change management request is basically a request for an operation to be performed.

The ChangeManager operates in terms of various change management operations. In turn, it interacts with managed environments via their touchpoints. The touchpoints perform a specified set of actions to carry out the change management operation (or a part of one) on behalf of the ChangeManager.

This chapter describes the change management operations defined by the Solution Installation architecture.

Note: The examples and descriptions in this chapter are based on our work with an early version of the Solution Installation for Autonomic Computing toolkit. Therefore, some differences may exist with the generally available version.

4.1 Operations

When you think of change management and the life cycle of software packages, there are several operations that one might need to perform. For instance, and in generic terms, install, configure, upgrade, reconfigure, uninstall.

The life cycle of a software package is generally defined by various states that the package can be in. Change management operations affect the state of the software package. The primary states of a software package are:

- Created
- Usable
- Updated

Note that different IUs within the same package can be in different states.

These states can roughly be defined as the following:

Created This is the typical state of an installable unit immediately after it has been installed. It may not be usable yet if there is configuration yet to be performed.

Updated An already installed IU may be updated. Additional configuration may need to be performed before it is again in a usable state.

Usable This is the state of the IU when it is in a state where it can be used. This would be the state after an IU has been created or updated and configured.

Note: The typical state after a create or updated operation may very well be usable. An IU will only be in the created state after a create operation if it contains an InitConfig artifact. It will only contain that artifact when special actions are required that have to be performed after the IU files are laid down.

We reference these states and state transitions as we describe the various change management operations below.

The change management operations supported by Solution Installation can be seen in the base.xsd schema definition file under %installdir%/common/acu/schemas. As can be seen, the following valid operations are defined:

- Create *
- Update *
- Undo *
- InitialConfig *

- Migrate *
- Configure
- VerifyConfig
- VerifyIU
- Repair
- Delete *

Restriction: In the current version of Solution Installation, only the operations above marked with an asterisk (*) are supported. Support for the other operations will be provided in future versions of the toolkit.

The schema definitions for these change management operations, as extracted and summarized from base.xsd, are shown below.

Example 4-1 Change management operations as defined in base.xsd

```
<!--##### IU LIFECYCLE OPERATIONS #######################-->
<simpleType name="Operation">
   <restriction base="NCName">
      <enumeration value="Create">
      </enumeration>
      <enumeration value="Update">
      </enumeration>
      <enumeration value="Undo">
      </enumeration>
      <enumeration value="InitialConfig">
      </enumeration>
      <enumeration value="Migrate">
      </enumeration>
      <enumeration value="Configure">
      </enumeration>
      <enumeration value="VerifyConfig">
      </enumeration>
      <enumeration value="VerifyIU">
      </enumeration>
      <enumeration value="Repair">
      </enumeration>
      <enumeration value="Delete">
      </enumeration>
   </restriction>
</simpleType>
```

It is up to each installable unit, through its descriptors and other artifact files, to define the checks, requirements, actions and so on that need to be handled for each management operation they support.

The expected behavior attributed to each operation is provided in a synopsis below. Operations typically affect the state of installable unit. Therefore, in the following descriptions you will see references to state transitions.

Create operation

The create operation typically is associated with the initial installation of an installable unit.

The create operation creates a new instance of an SIU. The Install artifact associated with the SIU defines the actions to be executed on the hosting environment to instantiate the SIU. Though the create operation creates an instance of the SIU on the target hosting environment, information about aggregate IUs and feature associations is stored in the IU and relationship registries.

The newly created IU instance transitions directly to the Usable state if no InitialConfig artifact is specified. Otherwise, the instance enters the Created state and an InitialConfig operation is needed to bring the instance to the Usable state.

On some hosting environments, the operation may be used to overwrite an existing instance of the SIU. The end result should be the same that would be obtained by performing the Create operation after applying the Delete operation to the existing instance.

An SIU may define a new base, an IU update, or a temporary fix. The Create operation can only be performed for an SIU defining a new base or a full update. An SIU defining a temporary fix or an incremental update can only be applied to an existing instance using the Update operation.

Update operation

The Update operation updates an existing IU instance. The SIU defining an update or temporary fix contains a declaration of the version range of a base IU instance onto which it can be applied (update base). The Install artifact associated to the SIU defines the actions to be executed on the hosting environment to update the base instance.

After the update, the version of the updated instance is changed to reflect the version specified by the SIU update. Version is *not* modified for fix packages.

The updated IU instance transitions directly to the Usable state if no Migrate artifact is specified. Otherwise, the instance enters the Updated state and the Migrate operation is needed to bring the instance to the Usable state.

The update may be applied in undo-able mode. In this case, any resources (for example, files) associated with the instance being updated that are being

replaced or modified need to be saved, in order to support the roll-back of the update. Not all update operations are undo-able. Only update operations for fix packages can be undone. And those can only be undone if the SIUs are all specified as undo-able and if all SIUs are installed on touchpoints that support undo-ability.

SIU updates and fixes may supersede fixes that are already applied to the instance being updated. Supersede is only supported for incremental updates at this time.

InitialConfig operation

This operation defines the configuration actions to be taken directly after creation.

The InitialConfig operation applies the initial configuration to an instance of the installable unit that is in the Created state, causing the transition to the Usable state. The operation can only be performed for an SIU defining a new base or a full update. The operation cannot be performed for an SIU defining a temporary fix or an incremental update.

The InitConfig artifact associated with the SIU defines actions to be executed on the hosting environment to make a created instance usable. This artifact implements the non-repeatable part of the initial configuration.

Migrate operation

The Migrate operation applies a configuration to an installable unit instance that is in the Updated state, causing the transition to the Usable state. The operation can only be performed for an SIU defining an update (full or incremental). The operation cannot be performed for a fix.

The Migrate artifact associated with the SIU defines actions to be executed on the hosting environment to make an updated instance usable. Actions in the artifact set the configuration by using existing installable unit instance data. The actions may implement the migration of configuration data used in the previous version within the instance being superseded. The Migrate artifact implements the non-repeatable part of the configuration process.

Configure operation

The Configure operation applies artifacts associated with configuration units that modify the configuration of a resource. This operation is not currently supported or implemented.

Delete operation

The delete operation deletes an existing IU instance from the hosting environment.

The Uninstall artifact associated with the SIU defines the actions to be executed on the hosting environment to remove resources (for example, files) created as part of the instance.

Some hosting environments may support using the install artifact for both the Create and the Delete operation. In that case, the install artifact is processed by the Delete operation to identify resources created during install that need to be removed.

The IU instance may have gone through multiple updates during its life cycle. The semantic of the Delete operation is that all resources created as part of the IU initial creation and subsequent updates should be removed, unless they already existed when the instance was first created (or updated), in which case they should not be removed.

VerifyIU operation

The VerifyIU operation is not currently supported or implemented.

VerifyConfig operation

The VerifyConfig operation is not currently supported or implemented.

Repair operation

The Repair operation is not currently supported or implemented.

4.2 Operations usage

The overall installation program will invoke one or more of the above change operations. For example, the provided sample installation program (ManageIU) takes as a parameter one of the operations. For a complete installation, you may need to invoke more than one operation. For instance, you might use a Create operation to perform the physical installation of a software package and then perform an InitialConfiguration operation to complete the configuration to ensure that the software package is ready to be started and used.

A more sophisticated installation or management program could choose to define its own set of operations that could wrap one or more of the Solution Installation operations. For example, it might define an Install operation that under the covers calls both the Create and InitialConfiguration operations.

Once the installation program passes the desired operation onto the ChangeManager, the descriptor and other artifacts will be properly parsed and/or utilized by other components such as the DependencyChecker based on the specific operation. Eventually the actions associated with the specific operation will be carried out by the touchpoint associated with the target hosting environment.

4.3 Summary

This short chapter has introduced the various change management operations defined by the Solution Installation architecture. One or more of these operations may be called to accomplish the desired state change for an installable unit.

These operations can be thought of as the macros that are supported by the ChangeManager. The ChangeManager and other Solution Installation components will parse and utilize the various descriptor and artifact files based on the request operation.

5

Touchpoints and hosting environments

This chapter focuses on the concept of touchpoints within the context of the Solution Installation. In short, touchpoints are the interfaces to the hosting environment that can be used by installation programs or other change management software.

In general, entities to be managed are called managed resources. Within the context of Solution Installation, managed resources are usually hosting environments. That is, environments that will host particular software or applications. Solution Installation provides the mechanisms to help manage the life-cycle of software. Software is installed, configured, upgraded, and uninstalled from a particular environment such as an operating system or a J2EE container. Other hosting environments can exist as well, for example, a DB2 environment might be considered a hosting environment where a change manager would want to be able to create, modify, and delete tables and other database objects such as stored procedures that might be part of an overall solution.

The touchpoint provides the interface that management software can use to invoke management operations on the managed resource.

The IBM Autonomic Computing architecture specifies that touchpoints should be implemented as Web services. In short, Web services are services designed to be usable by a variety of clients across a variety of environments. Web services

provide a basis for an open form of distributed computing where clients can dynamically locate the services they require, obtain information about the operations and data available from that service, and utilize the service.

Web services are defined through a standardized XML schema known as Web Services Definition Language (WSDL). WSDL define the contract between a service and potential client. Web services must implement the interfaces (known as portTypes) as expressed in the WSDL files that define the service. Likewise, writers of clients use the WDSDL files to ensure their request for a service conforms to the syntax (parameters and return values) of the service.

Note: It should be noted that touchpoints are described in this document as they relate to the Solution Installation technologies. However, touchpoints will provide a complete manageability interface. Refer to Figure 1-2 on page 4 for how touchpoints fit into the general autonomic computing architecture.

Note: The examples and descriptions in this chapter are based on our work with an early version of the Solution Installation for Autonomic Computing toolkit. Therefore, some differences may exist from the generally available version.

5.1 Hosting environments

As mentioned above, the two hosting environments that have been the focus of early work with the Solution Installation are the Operating System hosting environment and the J2EE container hosting environment. Other hosting environments can be envisioned as well.

For each hosting environment, there will need to be an implementation of a touchpoint Web service.

Currently, there are implementations of the Operating System touchpoint available for Windows and Linux.

From a J2EE Container perspective, the WebSphere Network Deployment facility implements the required interfaces and provides the touchpoint support for WebSphere Application Servers.

5.2 Touchpoint interfaces

Touchpoints are Web services that provide the interfaces for management applications to interact with a managed resource. In Web services terminology, sets of programming interfaces are known as portTypes. They are defined with Web Service Definition Language (WSDL) documents. WSDL documents are XML documents based on specific schemas.

WSDL documents contain two types of information:

- The portTypes that the touchpoints must implement. Each portType contains descriptions of operations, which can be performed on the manageable resource. For example, a RuntimeManagement portType might contain Start and Stop operations.
- Details of the bindings for this touchpoint. This section of the WSDL describes the various ways that a requesting system can access the touchpoint. For example, a touchpoint representing a managed resource might provide both a Web-based (SOAP/HTTP) and Java-based (RMI) interface—in this case, the WSDL would contain both SOAP and Java binding sections.

Note: Even though touchpoints are implemented as Web services, it is not necessary for them to provide a SOAP binding. If only local bindings are required, the interface WSDL need only include Java and/or C binding information.

In the current version of the Solution Installation toolkit, only local Java bindings are supported. Support for distributed protocols such as SOAP/HTTP will be made available in the future.

There are a variety of portTypes and they can be combined or composed somewhat similarly to the way that object-oriented programming interfaces can through interface inheritance. Therefore, touchpoints actually implement several portTypes, defined across a set of WSDL files, to provide their complete set of capabilities. There are some common operations that any touchpoint needs to implement, and there are others that may be specific to a hosting environment type (for example OS versus J2EE container).

In the discussion that follows, we highlight the primary portTypes that are implemented for the various touchpoints. The WSDL files that contain the actual definitions are located in <installdir>/IBM/common/acu/wsdl.

5.3 Touchpoint portTypes

Note: The information in this section may help cement some of the Solution Installation concepts in relationship to touchpoints. However, for most users of Solution Installation that will be generating descriptors and installation programs. The specifics of the WSDL are not important and not needed. The primary purpose for understanding the WSDL is for someone who may be generating a new touchpoint. In that case, they still would only need to understand the portTypes that must be implemented by all touchpoints and not the specifics of the operating system or WebSphere touchpoints as described below.

The WSDL files for most of the portTypes described in this section are located in the <installdir>/acu/wsdl directory.

5.3.1 portTypes for all touchpoints

There are several portTypes that are commonly implemented across all touchpoints.

Common portType

The common portType defined in common.wsdl defines the following operations:

- get
- set
- assignManager
- query
- setReferenceProperties

This portType provides a standard way to query and/or set properties of a managed resource.

Identification portType

The Identification portType makes a set of properties available that identify the managed resource. These properties include information such as a Machine Readable ID (MRID), host name, operating system type and operating system version.

Hosts portType

This portType is defined in the file hosts.wsdl and it defines interfaces for accessing resources associated with a specific host.

The operations defined by the Host portType are:

- updateHostedResource
- undoHostedResourceUpdate
- createHostedResource
- deleteHostedResource
- setHostedResourceInitialConfiguration
- commitHostedResourceUpdate
- verifyHostedResource
- repairHostedResource

HostingEnvironment portType

The HostingEnvironment portType provides interfaces for interacting with a generic hosting environment. It is defined in the file HostingEnvironment.wsdl.

The operations available by an implementation of this portType include:

- listAllInstalledSoftware
- listInstalledSoftware
- runProgram
- executeCheck

Specifically, the runProgram and executeCheck operations can be used to run custom checks or pre-defined checks on the target hosting environment to validate the environment before performing a change management operation.

Note: The term *operation* as used in the touchpoint context refers to a Web service operation, which can be thought of as a method or procedure call. Do not confuse this use of the term operation with the change management operations described in Chapter 4, “Solution Installation operations” on page 77.

5.3.2 Operating system touchpoint

The operating system specific touchpoint, in addition to the operations and data described above, provides access to a number of properties that may be useful for a change manager. Some of these include:

- IPAddress
- OSType
- Partition
- Processor
- SMBIOS
- Storage
- TotalVisibleMemorySize
- Version
- currentTimeZone
- distributed
- freePhysicalMemory
- freeSpaceInPagingFiles
- freeVitualMemory
- lastBootupTime
- localDateTime
- maxProcessMemorySize
- maxProcessesPerUser
- numberOfLicensedUsers
- numberOfProcesses
- sizeStoreInPagingFiles
- totalSwapSpaceSize
- totalVirtualMemorySize

Restriction: Not all of the above properties are accessible in the current version of the OS touchpoint. However, these and other properties will be made available in future releases.

5.3.3 WebSphere cell touchpoint

The WebSphere touchpoint is currently implemented and provided by the WebSphere Network Deployment feature. This model of having hosting environments implement their own touchpoints by providing a Web service interface to their management capabilities touchpoints will be extended to other hosting environments in the future.

By having built-in touchpoints that conform to the touchpoint Web service interfaces, hosting environments will be manageable by any management application conforming to the autonomic computing architecture.

In the case of the WebSphere touchpoint, though it is provided by WebSphere, it still needs to be registered with the Solution Installation environment. This registration is accomplished through a command (**`si_wstp_inst`**) provided with the Solution Installation toolkit.

In addition to the common portTypes described in 5.3.1, “portTypes for all touchpoints” on page 88, the WebSphere touchpoint implements three additional portTypes: J2EEDomain, WsManagement and WebSphereManageability.

We will not discuss the details of these touchpoints in this book, but through the aggregate set of portTypes implemented by the WebSphere touchpoint, the WebSphere Application Server is manageable using the autonomic architecture and specifically allows for the ChangeManager to use the touchpoint to perform change management operations.

There are three primary resource types that are valid for the WebSphere touchpoint.

DeployableObject These resources are typically the J2EE applications or their components (such as WebModules or EJBModules) and other artifacts associated with a WebSphere-based application.

DeploymentTarget This resource type represents one or more WebSphere Application Servers. These are the environments in which the WebSphere applications execute.

J2EEResource Another resource that may be accessed by the WebSphere application such as a JDBC data source, a JMS message queue or destination, and so on.

Like any touchpoint, there is a set of actions that would apply to a WebSphere touchpoint. The currently defined actions include:

- installJ2EEApp
- unInstallJ2EEApp

- wsadminAction

5.4 Summary

This chapter has provided a brief overview of touchpoints within the Solution Installation context. Touchpoints are implemented as Web services. Initially, the current toolkit provides local java bindings for these services, but over time the supported bindings will be expanded to support remote calls through such common standards as SOAP/HTTP.

The intention of this chapter was to provide basic concepts related to touchpoints and to give the reader a general understanding of their architecture and the types of interfaces (portTypes) they expose.

6

Creating an installation program

In this chapter, we cover the installation process and how to create an installation program that takes advantage of the Solution Installation framework.

The examples provided in this chapter are not intended to be comprehensive. They provide instructive information to allow the reader to get started on designing and writing their own installation or change management application. They provide specific and relatively simple examples of interfacing with the key components of the Solution Installation toolkit.

Note: The examples and descriptions in this chapter are based on our work with an early version of the Solution Installation for Autonomic Computing toolkit. Therefore, some differences may exist from the generally available version.

6.1 Basic flow

Before starting the installation process, let us review a few terms that are used in this chapter.

6.1.1 Common terms

Some common terms are:

Hosting environment This is the environment where a package is installed. It could be an operating system, a configured application server within a WebSphere cell, a database product (not supported by SI as of now), and so on.

Installation database This is a set of registries where information about the installed packages, their vendors, versions, deployment information, and so on is kept.

Checks Checks are used to obtain information about the target environment and help to set options for the installation program.

Requirements Requirements are the conditions out of which at least one must be satisfied so that an IU can be installed.

6.1.2 Installation process

Let's first look at the installation process and various steps involved in it. A complete installation process of a software package can be described broadly as involving the following steps. Note that the following steps highlight the tasks that need to be accomplished. However, as seen in the details presented throughout the rest of the chapter, many of the specific tasks are actually handled by the Solution Installation run time and are not the responsibility of the installation program developer. The details are outlined here for completeness and to help the reader understand the complete context.

1. Checking the environment as defined by the software package

   A software package has dependencies on the target hosting environment. These dependencies can include items such as available space for installation, available temporary space, number of processors, speed of processors, and so on. Based on the result of these checks, a user can be prompted for more information. Users should only provide input after the checks have been performed. This implies that all variables assigned within these checks in a deployment descriptor must have initial values. This type of

check is optional and may be omitted by the developer, depending upon the package. There are various types of checks to be performed, namely property, version, capacity, consumption, software and custom check.

2. Obtaining input from the user and validating that input

   The user may be able to select options related to the package, such as what features to install, target directories, and so on. Users can provide the input interactively or using a response file. There may be variables and elements in the deployment descriptor that are specific to the user's environment. They are not known at the time of creating the installation package. Hence, these values need to be overridden by user input. This input can be provided by users either interactively (choosing check boxes in a GUI or typing on a console) or using a response file. Once these values are known, the temporary values are replaced by the user input values. A few examples of such values might be the name of a database or IP address of a server.

3. Obtaining and validating the dependencies of the package

   This step looks for the prerequisites and ex-requisites. Prerequisites are the other software packages that must be installed before the current software package can be installed. Ex-requisites are software packages that should not be installed for the current package to be installed. Dependencies defined in the deployment descriptor are used by the dependency checker and verified against environment checks and by queries against the registry. Once the dependencies are found to be satisfied, the installer can continue the actual installation. This is a decision point within the execution of the installation program.

4. Installing the package

   Once the previous three steps have been completed and conditions indicate that the installation should proceed, then the actual installation of the package is initiated. This involves performing the actions as defined in the artifacts file (like copying files, creating directories, and so on). The install program calls APIs provided by Solution Installation, which triggers the actions to be performed through the touchpoints. The action artifacts defined in (or referenced by) the descriptor file are part of the install/uninstall process and are executed by the touchpoint of the corresponding hosting environment. These actions are invoked asynchronously. The installation program can be notified when a set of actions for an IU is complete through the ChangeListener interface. The action is complete and given information related to its success or failure. The installation program also has the ability to query the status of the IU and its associated actions while the operation is being carried out.

5. Registering the installable units in the package with the Solution Installation registry

Once the installation is complete, registration of the package must be done with the Solution Installation registries. Registration helps keep track of different versions of software installed in the environment. The Solution Installation registry not only maintains an inventory of what packages are installed, but also tracks the relationships among them. As will be seen, this registration is handled by the Solution Installation run time and not the actual installation program.

The above steps are represented by the figure below. Another important process not described above nor shown in the figure below is related to the generation and execution of an installation plan. This is handled internally by the run-time components, but is critical to the successful completion of operations involving complex packages containing multiple IUs that may have various dependencies on each other.

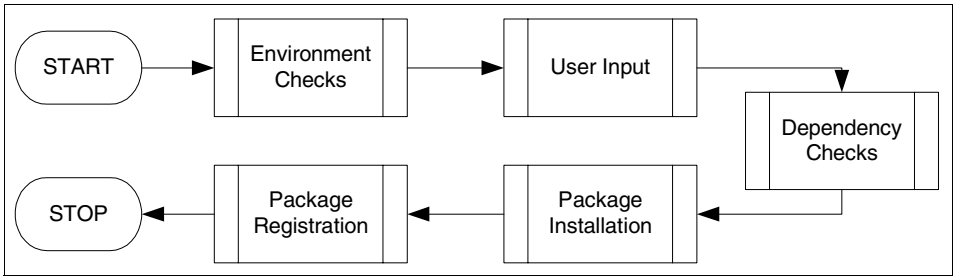

Figure 6-1 Installation process flow

6.2 Writing a custom installer

Throughout the rest of this chapter, it is highly recommended that the reader also refer to the javadoc files that come with the Solution Installation toolkit. These files provide the definitive reference for the Java classes and interfaces described in this chapter.

Based on the installation steps described above, let us create a simple installer that carries out these steps.

The first step is to gain access to the contents of the deployment descriptor and to perform other initialization steps.

Create a MediaInfo object using the deployment descriptor file location. The MediaInfo object is used for reading the deployment descriptor.

```
MediaInfo miObj = new MediaInfo(ddURI,null,null);
```

Where ddURI is the absolute/relative path of the deployment descriptor.

Now we have a stream for the deployment descriptor and need to parse it. Instantiate a DependencyChecker object using the DependencyCheckerFactory and use this object to parse the deployment descriptor.

```
DependencyChecker dc = DependencyCheckerFactory.createDependencyChecker();
IUDeploymentDescriptor iudd = dc.parseDeploymentDescriptor(miObj);
```

At this point, the DependencyChecker (dc) is created just for parsing the deployment descriptor. Actual dependency checking will be performed as seen throughout the steps below.

The next step is to create a RootIUTypeID. A RootIUTypeID is used for uniquely identifying the IU deployment descriptor.

```
RootIU rootIU = iudd.getRootIU();
root_typeID = (RootIUTypeID) IUTypeIDFactory.createIUTypeID(rootIU);
```

The following figure represents the initialization that we have accomplished so far. The output (parsed deployment descriptor) has only been partially parsed within our logic, but we have the ability to obtain information needed from the deployment descriptor through the IUDeploymentDescriptor (iudd) object.

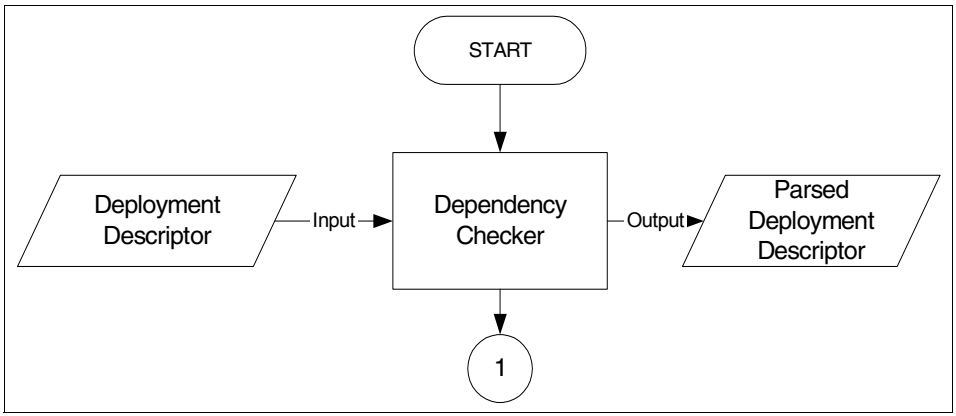

Figure 6-2 Initialization phase on installation program

6.2.1 Checking the environment as defined by the software package

The installation program must manage the checks that will be performed. By default, all checks are performed. However, based on the environment or user input, only a subset of checks may need to be performed. This section generically describes how to perform all of the checks defined in the deployment descriptor. Thus, to some extent it is creating logic that would be performed by default anyhow. The code snippets shown here can be used and modified by the

reader as a basis for more complex logic to manage the actual checks to be performed.

In this step we run elementary checks and see the results. For that, we will need the list of all checks to be performed in an array.

```
List allChecks = CheckerUtil.getAllChecks(rootIU);
```

The above method obtains all the desired checks from the deployment descriptor in a list. However, the DependencyChecker.elementaryCheck() method takes an array as an argument. The elementaryCheck() method will actually execute the checks. So we must convert the list to an array of Check objects.

```
Check[] checkArr = (Check[])allChecks.toArray(new Check[0]);
```

Now, we can call the elementaryCheck() method of the DependencyChecker object (dc, as created before). This method returns an array of CheckResult objects after performing the checks as listed in the deployment descriptor.

```
CheckResult[] checkResults = dc.elementaryCheck(iudd, checkArr, null,
null);
```

If the checkArr variable represents an empty array, then (as mentioned above) by default, all checks are performed.

As discussed in the previous section, there are various types of checks (property, software, and so on.) Depending on the type of check, different types of subclasses of objects are returned as CheckResult objects. But, there is one common attribute that CheckResult will return for all checks and that is status. Use the getStatus() method of CheckResult to see if the check was successful or not. This method returns a boolean value.

The getCheck() method of CheckResult will return a Check object. Use the getId() method of this Check object to get the ID of the check.

```
for (int i = 0; i < checkResults.length; i++) {
   CheckResult cr = (CheckResult)checkResults[i];
   boolean status = cr.getStatus();
   String id = cr.getCheck().getId();
}
```

Once it is determined that the check was successful, you may get the array of Selection objects from the successful check. Selection objects are used to retrieve the data returned by the check. For example, if the check was for operating system, the Selection object would return a single Property object. This Property object would contain the name and value of the property as defined in this check in the deployment descriptor. This object can be used to see what property was checked. The id can also be used to identify the check but Property is usually more descriptive than ID. For example, you may define your OS check

id as OS_check1, but the property would tell you that you actually want to check OSType and are expecting values such as Windows 2000.

Two sub-classes of Selection are primarily used, SoftwareSelection and PropertySelection. Depending upon the type of check, either a PropertySelection or SoftwareSelection object is returned. For property, version, capacity and consumption checks, a PropertySelection object is returned. For software checks, a SoftwareSelection object is returned. For custom checks, only status is returned.

The PropertySelection object contains the property that was queried and the result of the query from the hosting environment. Suppose you checked for physical disk space available on all drives on the target machine (property name in this case would be Partition). Then, the PropertySelection object would contain a list of Property objects that contains names and values of the property. In this case, each Property object would contain a value indicating the available disk space in a drive and the name as the name of the property (Partition). Corresponding to each drive (or file system, in case of UNIX®) present on the target machine, there will be a Property object in the PropertySelection object.

The code snippet given below checks if the Selection object is of type PropertySelection and then prints the name and value of each property.

```
Selection[] selections = cr.getSelections();
for (int j = 0; j < selections.length; j++) {
   Selection sel = selections[j];
   if (sel instanceof PropertySelection) {
      PropertySelection ps = (PropertySelection) sel;
      List listOfProps = ps.getProperties();
      Iterator it1= listOfProps.iterator();
      while (it1.hasNext())
         Property prop = (Property) it1.next();
         System.out.println("Property Name : " + prop.getName());
         System.out.println("Property Value : " + prop.getValue());
      } // end of while (it1)
   } // end of if (PropertySelection)
```

A SoftwareSelection object is returned when the check is of the type software. This class provides a list of IU instances that satisfied the query. Suppose you queried for software packages whose names start with IBM. Further, suppose that the query returns IBM Java Runtime, IBM WebSphere and IBM WSAD as results. Hence, the IU instance list would contain IUInstance objects corresponding to each result (a total of three instances in our example).

The code snippet below checks if the Selection object is of type SoftwareSelection and then prints the name, version and discriminant of each instance.

```
    if (sel instanceof SoftwareSelection) {
       SoftwareSelection ss = (SoftwareSelection) sel;
       List iuInstanceList = ss.getIUInstanceList();
       Iterator it2= iuInstanceList.iterator();
       while (it2.hasNext()) {
          IUInstance iu = (IUInstance) it2.next();
          System.out.println(" IU name: " + iu.getName() + " GUID: " +
iu.getGUID() + " Version: " + iu.getInstanceID().getVersion() + " Location:
" + iu.getInstanceID().getDiscriminant());
       } // end of while(it2)
    } // end of if (SoftwareSelection)
```

For simplicity, the sample code above just displays the results of the checks and values of successful checks. In a production-level installation program, additional logic might be included based on the results of these checks.

Figure 6-3 represents the dependency checker and its elementary checks as described above.

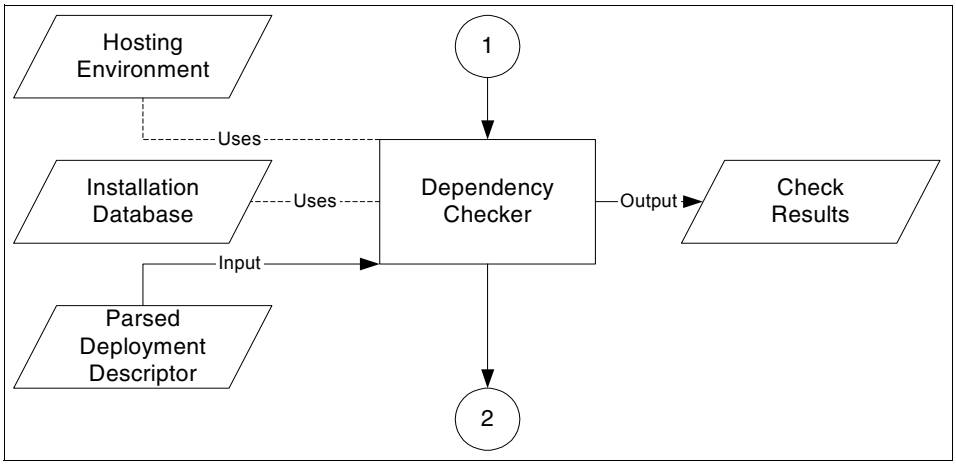

Figure 6-3 Performing environment checks

6.2.2 Obtaining user input and overriding default values

There may be variables in a package whose value may be provided by the individual performing the installation. Similarly, there are features that can be

selected to be installed on the target environment. In such conditions, users can provide input in two ways.

Interactive input Users can be prompted to provide input through an interactive user interface (like traditional installers created by InstallShield or a command line installer).

Response file A response file can be used to provide input data in a specified format that is understandable by the installer. This would typically be used in those cases where unattended installations are desired.

Choosing the most suitable option depends entirely on the requirements imposed by the environment and the needs of the business.

One point worth mentioning here is that when programmatically specifying values for variables or selecting features, the ID of the SIU is also required. The ID needs to be in one of the following forms:

1. SoftwareIUTypeID[*UUID,version*,RootIUTypeID[*UUID,version*]]
2. FeatureIUTypeID[*UUID,version*,RootIUTypeID[*UUID,version*]]
3. FixIUTypeID[*UUID,version*,RootIUTypeID[*UUID,version*]]
4. RootIUTypeID[*UUID,version*]

Where:

- *UUID* is the ID of corresponding element as declared in DD
- *version* is as mentioned in the DD for the element

If the descriptor contains the UUID of the software, feature, or fix type, then the RootIUTypeID is required as well. Though RootIUTypeID is also sufficient to identify the variable (or feature) selected, it is sometimes useful to fully qualify the variable (or feature) with the ID of the software, feature or fix. For those cases, use options 1, 2, or 3 above.

Note: IU descriptor data is case sensitive.

First, we will see how to override the value of a variable.

Suppose varName is the name of variable and varValue is the new value of the variable. Also, assume that descriptor is the string form of the UUID. The following sample code creates an IUTypeID and uses it to override a variable varName with new value varValue.

```
IUTypeID iud = (new IUTypeIDFactory()).createIUTypeID(descriptor);
ParameterOverrides po = new ParameterOverrides();
po.addOverride(iud,varName,varValue);
```

Note: varName must be equal to the variable name defined in the deployment descriptor and is case sensitive. If no match is found in the deployment descriptor for varName, it will be ignored.

For now, we will just maintain this variable (po) and move to feature selection. After completion of feature selection, we will use po again during the installation step.

Now let us see how to specify feature selection. As discussed above, we need the IU descriptor information. Let us assume that descriptor is the string form of UUID and featured is the ID of the feature. This ID is a String type and must be equal to the one defined in the deployment descriptor using the element:

```
<feature featureID="Feature_X">
```

Let us create the necessary data structure for feature selection.

```
RootIUTypeID rootIUTypeID = (RootIUTypeID)(new
IUTypeIDFactory()).createIUTypeID(descriptor);
SelectionID selId = new SelectionID(rootIUTypeID, featureId);
SelectionInput selInput = new FeatureSelectionInput(selId);
List featureList = new ArrayList();
featureList.add(selInput);
```

Now that we have variables representing the variables to be overridden (po) and the selected features (featureList) to be installed (as shown in Figure 6-4), we can move to the next step.

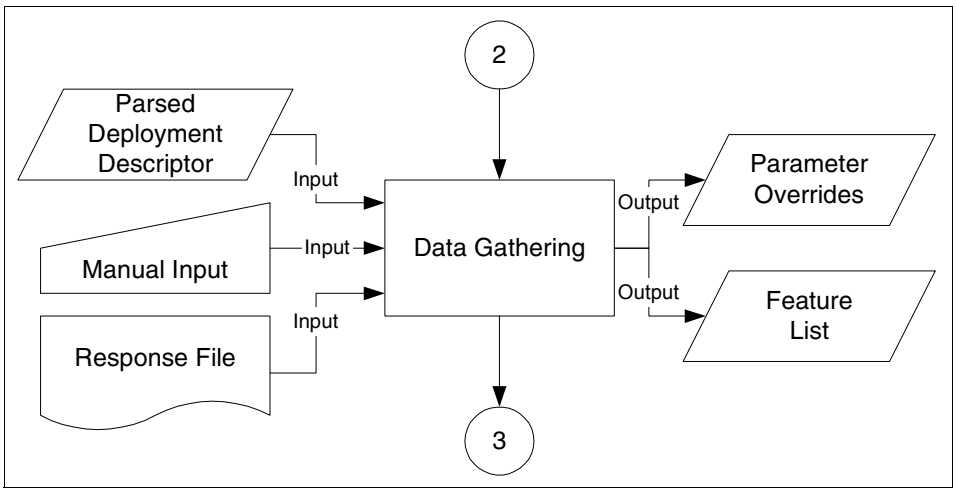

Figure 6-4 User Input

6.2.3 Obtaining and validating the dependencies of the package

Dependencies are defined in the deployment descriptor and available in the parsed deployment descriptor object (iudd).

Dependencies are checked against a change management operation. The operation could be create, delete, or update, for example. In this example, we are dealing with the create operation only. So, let us first create that operation.

A create operation needs an instance of the software to be installed. This software instance is of type SoftwareInstance and encapsulates the RootIU, name of RootIU, hosting environment, discriminant, and status of this software in the Solution Installation registry. The status can be deployable, created, usable, updated and so on. Possible states are listed in the javadoc for the SoftwareLifeCycle class. For our example, we would use SoftwareLifeCycle.STATUS_DEPLOYABLE. In general, when installing an IU for the first time, it starts out as deployable. After the initial create operation, the ChangeManager manages its state as different operations are performed on it. The SoftwareInstance object can then be retrieved from the registry and its state will be already set.

It should be noted that change management operations imply state changes, which in turn imply multiple states. We have not provided a detailed discussion here of states and state changes, but by referring to the javadoc referenced above, the reader can gain an understanding of the possible states and how transitions between theses states can occur.

Note: The discriminant is used for distinct identification of IUs of the same IU type, that is, to identify or separate one IU instance from another.

```
IUinstance ruiInst = new SoftwareInstance(root_typeID, rootIU.getIUName(),
"mrid:OSRT:Operating_System", discriminant,
SoftwareLifeCycle.STATUS_DEPLOYABLE);
ChangeManagementOperation cmop = new CreateOperation(ruiInst, null, false);
ChangeRequest changeRequest = new ChangeRequest(po, cmop,
null,featureList);
```

Note that in the above code fragment, we create an object of type CreateOperation. There are several related object classes all associated with different change management operations. These include DeleteOperation, and InstallOperation, MigrateOperation, and UndoOperation. They are all derived from an abstract class called ChangeManagement operation. Therefore, in your program you will need to create the appropriate object type based on the type of operation being performed.

It should be noted that for operations other than create, the SoftareInstance can be retrieved from the registry, rather than being created as shown in the first line of the above code fragment. To retrieve a SoftwareInstance from the registry, logic similar o the following can be used.

```
IURegistryView iurv = IURegistryFactory.createIURegistryView();
SoftwareInstanceData[] instanceArray =
   iurv.listSoftwareInstanceDataByUUID(root_typeID.getUUID(),
                                       null,
                                       null,
                                       null,
                                       null);
for (int i = 0; i < instanceArray.length; i++) {
   si = (SoftwareInstance) instanceArray[i].getIUInstance();
}// For our simple program/environment we assume only one
   // instance of an IU is installed.
```

Now we perform the actual dependency checking.

```
CheckDependenciesResult cdResult = dc.checkDependencies(iudd,
changeRequest, null, null);
```

The above statement invokes checking the dependencies defined in iudd for the specific ChangeRequest. After successful completion of the above method, results would come in the form of CheckDependenciesResult. The parameters, po and featureList, are declared and instantiated in the previous section as ParameterOverrides and List.

To check the results, we need to convert the results into ChangeManagementOperation`Result` objects. This can be done using the statement below.

```
ChangeManagementOperationResult cmor = cdResult.getCMOResult(root_typeID);
```

The cmor.getStatus() returns the status of the dependency check. A value of true means that the dependencies were satisfied.

If false, each child result of the cmor can be checked. The purpose of checking each child is to find out specifically which checks have failed. Child results can be checked, irrespective of whether `cmor.getStatus()` returns true or false.

```
ChangeManagementOperationResult[] childResults = cmor.getChildCMOResults();
```

You may check the status of each child and provide information to the user through messages or log files related to the reason the install could not continue if the requirements are not met.

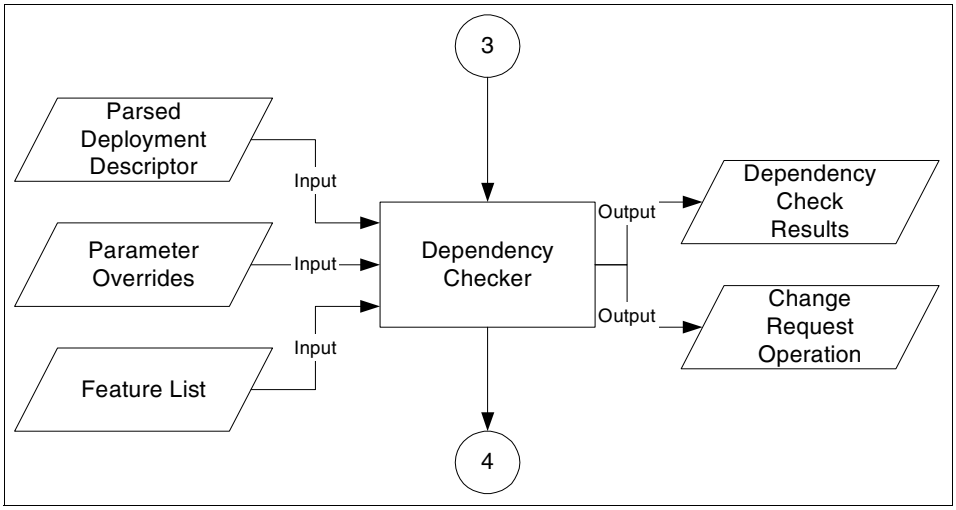

Figure 6-5 Dependency checks

6.2.4 Installing the package

Assuming that the dependencies have been met, the next step is to perform the actual installation. The ChangeManager object is responsible for interfacing with the touchpoint of the target environment and executing the actions defined for the change management operation.

```
IChangeManager chMgr = ChangeManagerFactory.createChangeManager();
ChangeID changeID = chMgr.change(iudd, mi, changeRequest, null, null, null,
this);
```

Either a MediaInfo, MediaReader or IUDeploymentDescriptor object needs to be passed. If both are null, an Exception will be thrown.

Note that the last parameter passed is *this*. This parameter is of type IChangeListener, which listens to the events generated by ChangeManager during the operation. Two methods, changeOccurred(ChangeEvent) and changeCompleted(ChangeCompleted), need to be implemented by the listener. In our example, since the program itself is the listener, these two methods need to be implemented. In some designs, a separate program could implement the IChangeListener interface.

changeOccurred() has one argument of type ChangeEvent. ChangeEvent has a few methods that can be used to obtain the status of the change. For simplicity, we could use getChangeRequestStateString(). ChangeCompleted extends ChangeEvent and hence the same getChangeRequestStateString() method can

be used for this event also. Sample code for the listener methods is shown below.

```
public void changeOccurred(ChangeEvent event)
{
   System.out.println(event.getChangeRequestStateString());
}

public void changeCompleted(ChangeCompleted event)
{
   System.out.println(event.getChangeRequestStateString());
}
```

The actions defined in the artifacts are executed by the touchpoint of the hosting environment.

Note that the installation program depends totally on the ChangeManager to handle the actual installation. The actions to be performed, based on the specific change management operation, are defined in the action artifacts and implemented in the touchpoint.

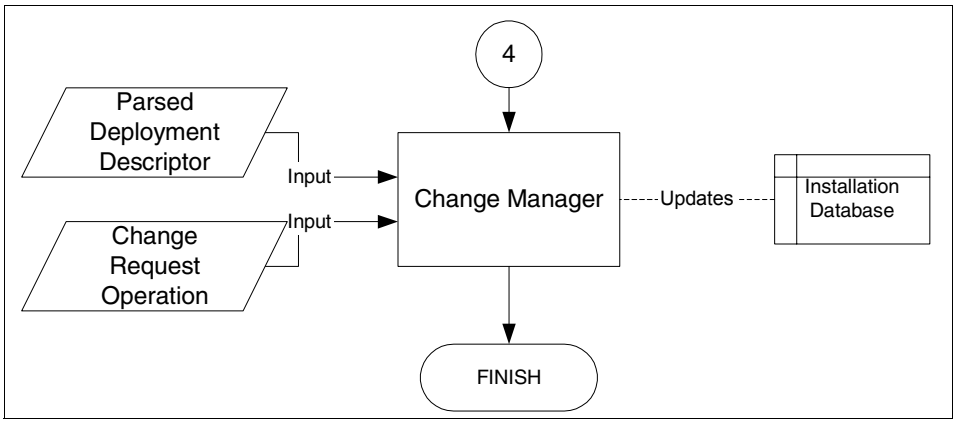

Figure 6-6 Operation execution

In the above diagram we do not show the fact that the ChangeManager utilizes other classes such as the DependencyChecker or show that the ChangeManager interacts with the target touchpoint.

6.2.5 Registering the installable units

Registration of the IU is done by the ChangeManager once the installation process completes. The installation program does not need to do anything for this. Relationships are also determined and registered by the ChangeManager.

6.3 Interfacing with ChangeManager and other classes

The Solution Installation APIs are provided through a variety of classes and their methods. These classes are used during various phases of the install cycle. A complete separate book can be written explaining all of the classes and their methods. Some of the important classes and their methods, which we will use for our install program, are discussed below.

Please refer to the javadoc provided with the toolkit for detailed information about these and other classes and interfaces available to the installation program.

6.3.1 com.ibm.ac.si.mr.MediaInfo

This class encapsulates methods to read data from various forms of packages. It is capable of reading the deployment descriptor and media descriptor from a local file system, from a removable media, within a zip file, from the network, and other forms. The constructor for the MediaInfo object is shown below:

```
public MediaInfo(String, String, String)
```

This is used to create a MediaInfo object passing a URI of either a media descriptor, deployment descriptor or package name.

6.3.2 com.ibm.ac.si.checker.DependencyChecker

DependencyChecker is primarily responsible for environmental checks and requirement checks. This includes property, version, consumption, capacity, software checks, prerequisite checks, topology checks, and so on.

- public IUDeploymentDescriptor parseDeploymentDescriptor(MediaInfo)

  Used to parse the deployment descriptor. The input argument is the MediaInfo object.

- public CheckResult[] elementaryCheck(IUDeploymentDescriptor, Check[], DependencyCheckListener, String[])

  Performs the elementary checks defined in the IUDD object. These checks are mainly property, capacity, consumption, software checks. A Check object represents a check defined in the deployment descriptor. DependencyCheckListener just has one method which is called when a dependency is checked. The String array contains the list of MRIDs of hosting environments against which the checks will be performed.

- public CheckDependenciesResult checkDependencies(IUDeploymentDescriptor, ChangeRequest)

Checks how the change management operation would affect the system. The ChangeRequest class is described 6.3.11, “com.ibm.ac.si.cm.ChangeRequest” on page 110.

6.3.3 com.ibm.ac.si.checker.CheckerUtil

This is a utility class for handling checks that are defined in the deployment descriptor.

```
public static List getAllChecks(EObject)
```

This method is used for getting the list of all preliminary checks defined in the deployment descriptor.

6.3.4 com.ibm.ac.si.checker.PropertySelection

This class represents the result of a check if the check is of type property, capacity, consumption or version.

```
public List getProperties()
```

This returns a list of properties if the check is successful. This list contains Property objects for all the matching results found. For example, if a capacity check is performed, the List would contain Property objects encapsulating capacities of all the matching objects.

6.3.5 com.ibm.ac.si.checker.SoftwareSelection

This class represents the result of a check when the check is of type software.

```
public List getIUInstanceList()
```

This returns a list of all the matching IU instances found. Class IUInstance has a few methods to get information about name, version, and discriminant of the IU.

6.3.6 com.ibm.ac.si.instances.ParameterOverrides

This class is used for capturing the variables in descriptors that are overridden.

```
public void addOverride(IUTypeID, String, String)
```

This adds a variable and its new value for overriding. The second parameter is the variable name and the third parameter is the new value. IUTypeID is the UUID of the IU that contains the variable.

6.3.7 com.ibm.ac.si.selections.FeatureSelectionInput

This class represents a selection. The ID of the feature is kept in the instance.

- public FeatureSelectionInput(SelectionID)

  The argument is the ID of the feature. This constructor conveys that the feature with this ID is selected and the sub-features are not selected. If you want to select the sub-features also, you may use the next constructor.

- public FeatureSelectionInput(SelectionID, boolean, boolean)

  This constructor is similar to above, with two additional parameters. The second parameter shows if the feature is selected or not. The third parameter does the same but for sub-features.

6.3.8 com.ibm.ac.si.instances.SoftwareInstance

This class represents an IU, CIU or SIU. The following is one of the constructors that we have used in our examples. There are other constructors also available.

```
public SoftwareInstance(RootIUTypeID, String, String, String, int)
```

RootIUTypeID is the not-null ID of RootIU. The next argument is the not-null name of the IU. The third argument is the non-null MRID of the hosting environment of the IU. The fourth argument is a nullable discriminant. The last argument indicates the status of the instance.

6.3.9 com.ibm.ac.si.cm.CreateOperation

CreateIOperation represents the change management operation of type ‘create’. This operation is used for installing a new instance of an IU, feature or fix.

```
public InstallOperation(IUInstance, IUTypeID, boolean)
```

The first argument is the non-null SoftwareInstance of the IU to be installed. The second argument is nullable and represents the IUTypeID of the instance that will be updated. The last argument is a boolean flag to represent if this operation can be undone.

6.3.10 com.ibm.ac.si.cm.DeleteOperation

The `DeleteOperation` represents the change management operation of type uninstall. This operation is used for uninstalling an instance of an IU, feature, or fix.

```
public DeleteOperation (IUInstance)
```

The argument is the non-null SoftwareInstance of the IU to be uninstalled.

6.3.11 com.ibm.ac.si.cm.ChangeRequest

This represents an object to build a change request.

```
public ChangeRequest(ParameterOverrides parms, ChangeManagementOperation cmop,
PhysicalTopology physicalTopology, java.util.List selections)
```

The first argument is the nullable ParameterOverrides object, which has all the overridden parameters and values. The second argument is the non-null ChangeManagement operation that is explained above. The third argument is nullable and represents the actual target hosting environment, and the fourth argument is the nullable list of selected features.

6.3.12 com.ibm.ac.si.cm.IChangeManager

This is an interface for calling change management operations. There are many constructors and methods in this interface. We use the following method for our samples.

```
public ChangeID change(IUDeploymentDescriptor, MediaInfo, ChangeRequest,
CheckDependencyResults, RequirementIdentifierList, ErrorHandlerOption,
IChangeListener)
```

This method is called to invoke the changes for the operation. The operation could be any of the valid operations. The first argument is the deployment descriptor object as explained above. The second object is the MediaInfo object that encapsulates the DD. Either of the first two parameters can be null. The third argument is described above. The fourth argument is nullable and if non-null, the same results will be used; otherwise new dependency checks will be run. The fourth argument is optional and represents requirements to ignore a check failure. The fifth argument is nullable and represents an error handler, and the last argument is the implementation of the listener that will receive notifications when the change management operation completes.

6.4 Sample installation program

Up to this point, we have provided various code snippets that implement key aspects of a simple installation program. The remainder of this chapter will describe a complete implementation of a very simple installation program.

Before we start creating our installer, let us describe our scenario. Our package will contain a deployment descriptor, an install artifact file, and an uninstall artifact file. In the payload, there is a text file that is to be copied to the hosting environment (operating system). There is just one IU in the deployment descriptor.

For the installation operation, a new folder will be created and the text file will be copied in to that folder.

For the uninstall operation, the file will be deleted and then the folder is also removed.

6.4.1 Manifest files

The following shows the deployment descriptor that we will use for our first example.

Example 6-1 packagedIU.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<iudd:rootIU
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:OSRT="http://www.ibm.com/namespaces/autonomic/OS_componentTypes"
xmlns:J2EERT="http://www.ibm.com/namespaces/autonomic/J2EE_RT"
xmlns:RDBRT="http://www.ibm.com/namespaces/autonomic/RDB_RT"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUD
D iudd.xsd"
IUName="SampleInstallerApplication_Single" targetRef="tOperatingSystem">

   <identity>
      <name>SampleInstallerRoot_Single</name>
      <UUID>12345432123454321234543212345432</UUID>
      <version>1.1</version>
   </identity>

   <installableUnit targetRef="tOperatingSystem">
      <SIU IUName="SampleInstallerAppSIU_ASingle"
hostingEnvType="OSRT:Operating_System">
         <identity>
            <name>SampleInstallerAppIU_ASingle</name>
            <UUID>0123456789abcdef0123456789abcdef</UUID>
            <version>1.0</version>
         </identity>
         <variables>
            <variable name="INSTALL_DIRECTORY">
               <parameter defaultValue="defaultASingle" />
            </variable>
         </variables>
         <unit>
            <installArtifacts>
               <installArtifact>
                  <fileIdRef>SampleInstallerActionsA</fileIdRef>
                  <parameterMaps>
```

```
                    <map>
                        <internalName>INSTALL_DIRECTORY</internalName>
                        <externalName>INSTALL_DIRECTORY</externalName>
                    </map>
                </parameterMaps>
            </installArtifact>
            <uninstallArtifact>
                <fileIdRef>SampleUnInstallerActionsA</fileIdRef>
                <parameterMaps>
                    <map>
                        <internalName>INSTALL_DIRECTORY</internalName>
                        <externalName>INSTALL_DIRECTORY</externalName>
                    </map>
                </parameterMaps>
            </uninstallArtifact>
        </installArtifacts>
    </unit>
</SIU>
</installableUnit>
<rootInfo>
    <schemaVersion>1.2.0</schemaVersion>
    <build>0</build>
</rootInfo>

<topology>
    <target id="tOperatingSystem" type="OSRT:Operating_System" >
        <selectionRequirements>
            <requirement name="OS">
                <alternative name="Windows_2000_Check_Alternative">
                    <inlineCheck>
                        <property checkId="Windows_2000_Check">
                            <propertyName>OSType</propertyName>
                            <value>Windows 2000</value>
                        </property>
                    </inlineCheck>
                </alternative>
                <alternative name="Windows_XP_Check_Alternative">
                    <inlineCheck>
                        <property checkId="Windows_XP_Check">
                            <propertyName>OSType</propertyName>
                            <value>Windows XP</value>
                        </property>
                    </inlineCheck>
                </alternative>
                <alternative name="Linux_Check">
                    <inlineCheck>
                        <property checkId="Linux_Check_Alternative">
                            <propertyName>OSType</propertyName>
                            <pattern>LINUX.*</pattern>
```

```
                    </property>
                 </inlineCheck>
              </alternative>
           </requirement>
        </selectionRequirements>
     </target>
  </topology>

  <files>
     <file id="SampleInstallerActionsA">
        <pathname>SampleInstallerActionsA.xml</pathname>
        <length>1407</length>
        <checksum />
     </file>
     <file id="SampleUnInstallerActionsA">
        <pathname>SampleUnInstallerActionsA.xml</pathname>
        <length>1303</length>
        <checksum />
     </file>
     <file id="testfile_A">
        <pathname>../FILES/SampleInstallerFile_A.txt</pathname>
        <length>56</length>
        <checksum />
     </file>
  </files>
</iudd:rootIU>
```

As you can see from the file, there is a single IU, namely “SampleInstallerApplication_Single”. The install artifact file is SampleInstallerActionsA.xml and the uninstall artifact file is SampleUnInstallerActionsA.xml. These are shown below.

Example 6-2 SampleInstallerActionsA.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<action:artifact
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:osac="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:rtype="http://www.ibm.com/namespaces/autonomic/solutioninstall/ResourceTy
pes"
xmlns:sigt="http://www.ibm.com/namespaces/autonomic/solutioninstall/Signatures"
xmlns:vsn="http://www.ibm.com/namespaces/autonomic/common/version"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsA
ctions osActions.xsd">
```

```
   <artifactType>Install</artifactType>
   <artifactSchemaVersion>1.2.0</artifactSchemaVersion>
   <actionGroup xsi:type="osac:OsActionGroup">
   <actions>
      <addDirectory>
         <directory>
            <location>./</location>
            <name>$(INSTALL_DIRECTORY)</name>
         </directory>
      </addDirectory>

      <addFile>
         <file>
            <location>./$(INSTALL_DIRECTORY)/</location>
            <name>SampleInstallerFile_A.txt</name>
            <source>
               <fileIdRef>testfile_A</fileIdRef>
            </source>
         </file>
      </addFile>
   </actions>
   </actionGroup>
</action:artifact>
```

The install artifact file has two actions. The first one creates a directory (INSTALL_DIRECTORY) and the second one copies the text file (referenced as testfile_A) into that directory.

Finally, here is the uninstall artifact file.

Example 6-3 SampleUnInstallerActionsA.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<action:artifact
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:osac="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:rtype="http://www.ibm.com/namespaces/autonomic/solutioninstall/ResourceTy
pes"
xmlns:sigt="http://www.ibm.com/namespaces/autonomic/solutioninstall/Signatures"
xmlns:vsn="http://www.ibm.com/namespaces/autonomic/common/version"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsA
ctions osActions.xsd">
   <artifactType>Uninstall</artifactType>
   <artifactSchemaVersion>1.2.0</artifactSchemaVersion>
   <actionGroup xsi:type="osac:OsActionGroup">
```

```
      <actions>
         <removeFile>
            <location>./$(INSTALL_DIRECTORY)</location>
            <name>SampleInstallerFile_A.txt</name>
         </removeFile>
         <removeDirectory>
            <location>./</location>
            <name>$(INSTALL_DIRECTORY)</name>
         </removeDirectory>
      </actions>
   </actionGroup>
</action:artifact>
```

The same two actions as seen in the install actions artifact are reversed in the uninstall operation. First, the text file is deleted and then the directory is removed. The directory name is referenced by $(INSTALL_DIRECTORY).

6.4.2 Installation program

The installation program is shown below. This is based on the information and examples we provided in the previous sections of this chapter.

Example 6-4 SampleInstaller.java

```
package com.itso.ac.si;

// Java imports
import java.util.*;
import java.io.*;

// Toolkit imports
import com.ibm.ac.si.checker.*;
import com.ibm.ac.si.cm.*;
import com.ibm.ac.si.iuregistry.*;
import com.ibm.ac.si.instances.*;
import com.ibm.ac.si.mr.MediaInfo;
import com.ibm.ac.si.selections.FeatureSelectionInput;
import com.ibm.ac.si.selections.SelectionID;
import com.ibm.ac.si.selections.SelectionInput;
import com.ibm.namespaces.autonomic.solutioninstall.iudd.RootIU;
import com.ibm.namespaces.autonomic.solutioninstall.base.IUIdentity;
import com.ibm.namespaces.autonomic.solutioninstall.base.Check;
import org.eclipse.emf.common.util.URI;

/** SampleInstaller class - A simple implementation of a program to take
advantage of the Solution Install toolkit to perform package creates and
deletes.
*/
```

```
public class SampleInstaller implements IChangeListener {
   private List featureSelections= null;
   private List deleteFeatures = null;
   private String operation = null;
   private String discriminant = null;
   private String responseFile = null;
   private String deployDescriptor = null;

/** Method main() */
   public static void main(String args[]){
      System.out.println("****** Sample Installer ******");
      System.out.println("Simple program to demonstrate ");
      System.out.println("some features of the");
      System.out.println("Solution Installation for");
      System.out.println("Autonomic Computing toolkit");
      System.out.println("******************************\n");
      if (args.length < 3){
         System.out.println("Complete data is not provided. Please rerun with
deployment descriptor, responseFile and operation");
         System.exit(-1);
      }
      String deployDescriptor = args[0];
      String responseFile= args[1];
      String operation= args[2] ;
      String discriminant= "ITSOSample";
      // if an optional fourth parameter is provided, it would be discriminant
      if (args.length == 4)
         discriminant = args[3];
      SampleInstaller sample = new SampleInstaller(operation, discriminant,
responseFile, deployDescriptor);
      sample.invokeOp();
   } // End main()

/** changeOccurred() is a callback method for the ChangeListener interface */
    public void changeOccurred(ChangeEvent event){
    System.out.println("SampleInstaller has recieved a changeOccurred Event: "
+ event.getChangeRequestStateString());
    } // End changeOccurred()

/** changeCompleted is a callback method for the ChangeListener interface */
    public void changeCompleted(ChangeCompleted event){
    System.out.println("SampleInstaller has recieved a changeCompleted Event: "
+ event.getChangeRequestStateString());
    } //End changeCompleted()

/**
 * Constructor for SampleInstaller
 */
```

```
   public SampleInstaller(String chMgrOp, String discriminant, String
responseFile, String deployDescriptor) {
      this.operation = chMgrOp;
      this.discriminant = discriminant;
      this.responseFile = responseFile;
      this.deployDescriptor = deployDescriptor;
   } // End constructor

/**
 * Private invokeOp() method:
 * Parses the deployment descriptor
 * Performs environment checks
 * Performs dependency checks
 * Performs ChangeManager operation
 */
   private void invokeOp(){
      SoftwareInstance si = null;
      try {
   //Begin initialization of key variables/objects
         MediaInfo mi = new MediaInfo(deployDescriptor, null, null);
         DependencyChecker checker =
DependencyCheckerFactory.createDependencyChecker();
         IUDeploymentDescriptor iudd = checker.parseDeploymentDescriptor(mi);
         RootIU rootIU = iudd.getRootIU();
         if (rootIU == null) {
            System.out.println("rootIU is NULL in the Deployment Descriptor.
Please check the DD." );
               System.exit(1);
         } //end if
         RootIUTypeID root_typeID = (RootIUTypeID)
IUTypeIDFactory.createIUTypeID(rootIU);
   // end initialization

   // Begin checks
         //Get all of the checks in the IUDD
         Check[] checkArr = null;  //If null array, all checks are performed
by default in next step
         // Run elementaryCheck and display the results
         CheckResult[] checkResults = checker.elementaryCheck(iudd, checkArr,
null, null);
         for (int i = 0; i < checkResults.length; i++) {
            CheckResult cr = (CheckResult)checkResults[i];
            System.out.println("\nCheck name:" + cr.getCheck().getCheckId());
            System.out.println(" Check status: " + cr.getStatus());
            Selection[] selections = cr.getSelections();
            for (int j = 0; j < selections.length; j++) {
               Selection sel = selections[j];
               // if the check is related to the consumption, property,
capacity and version
```

```
                if (sel instanceof PropertySelection) {
                   PropertySelection ps = (PropertySelection) sel;
                   List listOfProps = ps.getProperties();
                   Iterator it1 = listOfProps.iterator();
                   while (it1.hasNext()) {
                      Property prop = (Property) it1.next();
                      System.out.println("\nProperty name : " +
prop.getName());
                      System.out.println("Property value : " +
prop.getValue());
                   } // end while
                } else {
                   // if the check is related to the software
                   if (sel instanceof SoftwareSelection) {
                      SoftwareSelection ss = (SoftwareSelection) sel;
                      List iuInstanceList = ss.getIUInstanceList();
                      Iterator it2 = iuInstanceList.iterator();
                      while( it2.hasNext()) {
                         IUInstance iu = (IUInstance) it2.next();
                         System.out.println(
                         " IU name: "
                         + iu.getName()
                         + " Version: "
                         + iu.getInstanceID().getVersion()
                         + " Location: "
                         + iu.getInstanceID().getDiscriminant());
                      } // end while
                   } // end if
                } // end else
             } // end for j
          } // end for i
   // end of checks

   // begin get user input and feature selection
          ParameterOverrides po = null;
          if(responseFile!=null) {
             Properties respFileProps = new Properties();
             try {
                respFileProps.load(new FileInputStream(responseFile));
             }catch (IOException e){
                System.out.println("Error opening the response file.
Exiting...");
                e.printStackTrace();
                System.exit(1);
             } //end try/catch
             po = getParameterOverrides(respFileProps,iudd);
             featureSelections = getFeatureSelections(respFileProps);
          } // end if
   // end get user input and feature selection
```

```
   // begin operation processing
         ChangeManagementOperationcmop = null;
         IUInstance      riuInst = null;
         IUTargetResult targetResult = null;
         String             mrid = null;
         String[]        targetResultList = null;

          targetResult = checker.targetIU(iudd, root_typeID, false, null);
         targetResultList = targetResult.getHostingEnvironments();
         for (int i = 0; i < targetResultList.length; i++) {
            mrid = targetResultList[i];
            System.out.println("\nHosting Environment found is: " + mrid);
         }

         ChangeRequest changeRequest = null;
         // Build the CMOP for the Change Request
         if(operation.equalsIgnoreCase("create")) {
            System.out.println("\nBegin Create Operation");
            riuInst = new SoftwareInstance(root_typeID,
rootIU.getIdentity().getName(), mrid, discriminant,
SoftwareLifeCycle.STATUS_DEPLOYABLE);
            cmop = new CreateOperation(riuInst, null, false);
         changeRequest = new ChangeRequest(po, cmop, null, featureSelections);
         } else
            if(operation.equalsIgnoreCase("delete")) {
            System.out.println("\nBegin Delete Operation");
            IURegistryView iurv = IURegistryFactory.createIURegistryView();
            SoftwareInstanceData[] instanceArray =
               iurv.listSoftwareInstanceDataByUUID(root_typeID.getUUID(),
                                                   null,
                                                   null,
                                                   null,
                                                   null);

            for (int i = 0; i < instanceArray.length; i++) {
               si = (SoftwareInstance) instanceArray[i].getIUInstance();
            } // For our simple program/environment we assume only one
              // instance of an IU is installed.
            cmop = new DeleteOperation(si);
            // null for last parameter below dictate root and all features to
be deleted.
            changeRequest = new ChangeRequest(po, cmop, null, null); //null
         } //end if
         //Run check dependencies to find an alternative that meets the
requirements of the install operation.
         CheckDependenciesResult checkDependenciesResultList =
checker.checkDependencies(iudd, changeRequest, null, null);
```

```
         ChangeManagementOperationResult cmResults =
checkDependenciesResultList.getCMOResult(root_typeID);
         boolean clear = checkResults(cmResults);

         // If it does not meet the requirement, exit, otherwise, call
changeManager to do installation
         if (!clear){
            System.out.println("Requirements for operation were not met.
Exiting program.");
            System.exit(1);
         } // end if
         IChangeManager cm = ChangeManagerFactory.createChangeManager();
         ChangeID cID = null;
         cID = cm.change(iudd, mi, changeRequest, null, null, null, this);
      } catch (Exception e) {
         System.out.println("\nError detected from the Change Manager Change
method. Exiting.");
         e.printStackTrace();
         System.exit(1);
      } // end try/catch
   // end operation processing
   } // end invokeOp()

/**
* This recursive function checks cmo results
* If cmor.getStatus() returns true, it means we can go ahead doing
installation.
* Otherwise, get RequirmentResult, if the resultList is bigger than zero, print
out the list
* then go through each child of the cmo Results, to make recursive call to its
children
*/
    private boolean checkResults(ChangeManagementOperationResult cmor)  {
      RequirementsResultsrequirementsList =  null;
      boolean   result = false;

      if (cmor.getStatus()) {
         result = true;
      } else {
         requirementsList = cmor.getRequirementsResults();
         if (requirementsList.getRequirementResults().length > 0) {
            ChangeManagementOperationResult[] children =
cmor.getChildCMOResults();
            for(int i = 0; i < children.length; i++) {
               result = checkResults(children[i]);
            } // end for
         } // end if
      } // end if/else
      return result;
```

```
    } // End checkResults()

/**
* Gets a list of ParameterOverrides from the Properties object
*/
    private ParameterOverrides getParameterOverrides(Properties
props,IUDeploymentDescriptor iudd) {
       Enumeration list = null;
       String       name   = null;
       String       varValue = null;
       StringTokenizertoken = null;
       IUTypeID  iud    = null;
       String       descriptor = null;
       String       varName= null;
       String       temp   = null;
       ParameterOverridespo = new ParameterOverrides();
       final String VARIABLE = "Variable";

       list = props.propertyNames();
         System.out.println("\nParameter Overrides: ");
       while (list.hasMoreElements()) {
          name = (String)list.nextElement();
          varValue = props.getProperty(name);
          token = new StringTokenizer(name,"#");
          temp=token.nextToken();
          if (temp.equals(VARIABLE)){
             descriptor = token.nextToken();
             varName = token.nextToken();
             System.out.println(" Variable : " + varName);
             System.out.println(" Value : " + varValue + "\n");
             iud = IUTypeIDFactory.createIUTypeID(descriptor);
             po.addOverride(iud,varName,varValue);
          } // end if
       } // end while
       return po;
    } // End getParameterOverrides()

    private List getFeatureSelections(Properties props) {
       ListIts a new trend in technology basd featureList = new ArrayList();
       Enumerationlist = null;
       String    name      = null;
       String    val          = null;
       String    descriptor= null;
       String    featureId= null;
       String    temp      = null;
       StringTokenizertoken= null;
       RootIUTypeID rootIUTypeID= null;
       final String SELECTED_FEATURE = "Feature";
```

```
      list = props.propertyNames();
      while (list.hasMoreElements()) {
         name = (String)list.nextElement();
         val = props.getProperty(name);
         token = new StringTokenizer(name,"#");
         temp = token.nextToken();
         if (temp.startsWith(SELECTED_FEATURE)){
            descriptor = token.nextToken();
            featureId = token.nextToken();
            System.out.print("\n" + SELECTED_FEATURE + "Id : " + featureId);
            rootIUTypeID =
(RootIUTypeID)IUTypeIDFactory.createIUTypeID(descriptor);
            SelectionID selId = new SelectionID(rootIUTypeID, featureId);
            SelectionInput selInput = new FeatureSelectionInput(selId);
            featureList.add(selInput);
         } // end if
      } // end while
      return featureList;
   } // End getFeatureSelections()
} //End SampleInstaller class
```

6.4.3 Setup of installation program

To see this installation program in action, perform the following steps.

1. Create a folder named C:\Sample.
2. Go to C:\Sample and create a subfolder META-INF.
3. Similarly, create another subfolder FILES under C:\Sample.
4. Create another set of subfolders under C:\sample as com\itso\ac\si.
5. The deployment descriptor, install and uninstall artifact files should be created in (or copied to) the META-INF folder.
6. The sample text file should be create/stored under FILES.
7. Create a blank text file named Response.properties. This file will be used later, but to satisfy the installation program, we are creating an empty file for now.

   Note that the source files for this example are available for download. Please see Appendix A, “Additional material” on page 173, for information related to obtaining these files.

   Run <installdir>\setenv.cmd to set up the environment. <installdir> is the directory where the Solution Installation toolkit is installed. Typically, on a Windows platform it is C:\Program Files\IBM\Common\ACU.
8. Compile the source using javac com\itso\ac\si\SampleInstaller.java.

Payload

There is just one payload file for this sample. The sample text file is created as below.

Example 6-5 SampleInstallerFile_A.txt

```
This is a sample file for demonstrating SampleInstaller.
```

There is just a single line in the file for demonstration. Of course, for a real product, there would typically be many files of different types.

Response file

The response file can be blank for our example. As explained in the setup of the installation program, create an empty text file named Response.properties in C:\Sample.

Execution

Now, to run the sample, use the following command.

```
java com.itso.ac.si.SampleInstaller META-INF\single_packagedIU.xml
Response.properties create
```

This command starts the installer and points at our deployment descriptor, the response file that could contain variable overrides and passes the type of change management operation to perform (create).

Output

The following examples show the output of various commands related to running the SampleInstaller for the create and delete operations.

Example 6-6 Initial state

```
C:\Sample>dir
 Volume in drive C has no label.
 Volume Serial Number is 5CC0-9DEE

 Directory of C:\Sample

07/07/2004  09:08 AM    <DIR>          .
07/07/2004  09:08 AM    <DIR>          ..
07/07/2004  09:00 AM    <DIR>          FILES
07/07/2004  09:00 AM    <DIR>          Meta-INF
07/07/2004  09:09 AM                 0 Response.properties
               1 File(s)              0 bytes
               4 Dir(s)  48,884,080,640 bytes free
```

The above example shows a simple **dir** command that reflects the state of the sample directory.

Next we execute the create operation as shown below.

Example 6-7 Output from create operation

```
C:\Sample>java com.itso.ac.si.SampleInstaller Meta-INF/packagedIU.xml
Response.properties create
****** Sample Installer ******
Simple program to demonstrate
some features of the
Solution Installation for
Autonomic Computing toolkit
******************************


Check name:Windows_2000_Check
 Check status: false

Check name:Windows_XP_Check
 Check status: true

Property name : OSType
Property value : Windows XP

Check name:Linux_Check_Alternative
 Check status: false

Parameter Overrides:

Hosting Environment found is:
mrid:http://w3.ibm.com/namespaces/2003/OS_componen
tTypes:Operating_System

Begin Create Operation
SampleInstaller has recieved a changeOccurred Event: SUBMITTED
SampleInstaller has recieved a changeOccurred Event: EXECUTING
SampleInstaller has recieved a changeCompleted Event: COMPLETED
```

After the apparently successful operation above, we can use a few simple commands to see that the target directory (defaultASingle) has been created and the appropriate file (SampleInstallerFile_A.txt) copied to that directory.

Example 6-8 Validating installation

```
C:\Sample>dir
 Volume in drive C has no label.
 Volume Serial Number is 5CC0-9DEE
```

```
 Directory of C:\Sample

07/07/2004  09:10 AM    <DIR>          .
07/07/2004  09:10 AM    <DIR>          ..
07/07/2004  09:10 AM    <DIR>          defaultASingle
07/07/2004  09:00 AM    <DIR>          FILES
07/07/2004  09:00 AM    <DIR>          Meta-INF
07/07/2004  09:09 AM                 0 Response.properties
               1 File(s)              0 bytes
               5 Dir(s)  48,883,945,472 bytes free

C:\Sample>dir .\defaultASingle
 Volume in drive C has no label.
 Volume Serial Number is 5CC0-9DEE

 Directory of C:\Sample\defaultASingle

07/07/2004  09:10 AM    <DIR>          .
07/07/2004  09:10 AM    <DIR>          ..
07/07/2004  09:10 AM                56 SampleInstallerFile_A.txt
               1 File(s)             56 bytes
               2 Dir(s)  48,883,945,472 bytes free
```

We can also use the listIU utility supplied with the Solution Installation toolkit to verify the installation has been registered with the installation database, as shown in the following example.

Example 6-9 Using listIU utility to verify registration

```
C:\Sample>listiu
IU      UUID: 0123456789abcdef0123456789abcdef  Name:
SampleInstallerAppIU_ASingle      Version: 1.0
IU      RootIU UUID: 12345432123454321234543212345432   Name:
SampleInstallerRoot_Single        Version: 1.1
```

Now let us continue on by uninstalling (or deleting) the software package using the delete operation.

Example 6-10 Output from delete operation

```
C:\Sample>java com.itso.ac.si.SampleInstaller Meta-INF/packagedIU.xml
Response.p
roperties delete
****** Sample Installer ******
Simple program to demonstrate
some features of the
Solution Installation for
```

```
Autonomic Computing toolkit
******************************


Check name:Windows_2000_Check
 Check status: false

Check name:Windows_XP_Check
 Check status: true

Property name : OSType
Property value : Windows XP

Check name:Linux_Check_Alternative
 Check status: false

Parameter Overrides:

Hosting Environment found is:
mrid:http://w3.ibm.com/namespaces/2003/OS_componen
tTypes:Operating_System

Begin Delete Operation
SampleInstaller has recieved a changeOccurred Event: SUBMITTED
SampleInstaller has recieved a changeOccurred Event: EXECUTING
SampleInstaller has recieved a changeCompleted Event: COMPLETED
```

Again, we can verify its success by checking to see if the installation directory has been successfully removed as show below. Notice that the defaultASingle directory no longer exists.

Example 6-11 Checking to see if installation directory has been removed

```
C:\Sample>dir
 Volume in drive C has no label.
 Volume Serial Number is 5CC0-9DEE

 Directory of C:\Sample

07/07/2004  09:14 AM    <DIR>          .
07/07/2004  09:14 AM    <DIR>          ..
07/07/2004  09:00 AM    <DIR>          FILES
07/07/2004  09:00 AM    <DIR>          Meta-INF
07/07/2004  09:09 AM                 0 Response.properties
               1 File(s)              0 bytes
               4 Dir(s)  48,883,826,688 bytes free
```

Now, if we use the listIU utility, we see that the information about the software package has also been removed from the installation database.

Example 6-12 Using listIu after delete operation

```
C:\Sample>listiu
ACUIR00002I List is empty!
```

Important: The above sample has been demonstrated in a Windows environment. The same sample can be run on Linux also. However, the folder names would change slightly, for instance, instead of C:\sample, we may have /home/sample. Similarly, the **`dir`** command would be replaced with `ls`.

All the files (deployment descriptor, SampleInstaller.java, SampleInstaller.class, install/uninstall artifact files, Response.properties and sample text file) are provided in a zip that can be downloaded from the Internet. Please refer to Appendix A, “Additional material” on page 173, for information about obtaining the sample files.

Attention: The Response.properties in the sample zip file has one property. This is the variable that is used for overriding the value of INSTALL_DIRECTORY in the deployment descriptor and the artifact files. The installer will take care to override the value. If you do not want to override and want to use the default value, either remove that line or just comment it using a hash (#) in front of it.

6.5 Feature selection example

The above example made use of variable overriding. We now provide a modification to demonstrate the use of feature selection. The installation program would remain the same, but the deployment descriptor, artifact files, and response file would change.

Our scenario for this example contains a package that contains a deployment descriptor. This deployment descriptor would have two features and the package would also have an install artifact file and an uninstall artifact file for each feature. In the payload, there are two text files, once again one for each feature, that are to be copied to the hosting environment (operating system). So, there are two IUs in the deployment descriptor.

For the installation operation, a new folder is created and the text file is copied to the folder for the selected feature.

6.5.1 Manifest files

There are some changes in the deployment descriptor as shown below.

Example 6-13 packagedIU.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<iudd:rootIU
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:OSRT="http://www.ibm.com/namespaces/autonomic/OS_componentTypes"
xmlns:J2EERT="http://www.ibm.com/namespaces/autonomic/J2EE_RT"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUD
D C:\PROGRA~1\IBM\common\ACU\schema\iudd.xsd"
IUName="SampleInstallerApplication_Multiple" targetRef="tOperatingSystem">

   <identity>
      <name>SampleInstallerRoot_Multiple</name>
      <UUID>11112222333344445555666677778888</UUID>
      <version>1.1</version>
   </identity>

   <selectableContent>
   <installableUnit targetRef="tOperatingSystem">
      <SIU IUName="SampleInstallerAppSIU_AMultiple"
hostingEnvType="OSRT:Operating_System">
         <identity>
            <name>SampleInstallerAppIU_AMult</name>
            <UUID>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</UUID>
            <version>1.0</version>
         </identity>
         <variables>
            <variable name="INSTALL_DIRECTORY_A">
               <parameter defaultValue="defaultA" />
            </variable>
         </variables>
         <unit>
            <installArtifacts>
               <installArtifact>
                  <fileIdRef>SampleInstallerActionsA</fileIdRef>
                  <parameterMaps>
                     <map>
                        <internalName>INSTALL_DIRECTORY_A</internalName>
                        <externalName>INSTALL_DIRECTORY_A</externalName>
                     </map>
                  </parameterMaps>
               </installArtifact>
               <uninstallArtifact>
                  <fileIdRef>SampleUnInstallerActionsA</fileIdRef>
```

```
                    <parameterMaps>
                       <map>
                          <internalName>INSTALL_DIRECTORY_A</internalName>
                          <externalName>INSTALL_DIRECTORY_A</externalName>
                       </map>
                    </parameterMaps>
                 </uninstallArtifact>
              </installArtifacts>
           </unit>
        </SIU>
     </installableUnit>
     <installableUnit targetRef="tOperatingSystem">
        <SIU IUName="SampleInstallerAppSIU_BMultiple"
  hostingEnvType="OSRT:Operating_System">
           <identity>
              <name>SampleInstallerAppIU_BMult</name>
              <UUID>BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB</UUID>
              <version>1.0</version>
           </identity>
           <variables>
              <variable name="INSTALL_DIRECTORY_B">
                 <parameter defaultValue="defaultB" />
              </variable>
           </variables>
           <unit>
              <installArtifacts>
                 <installArtifact>
                    <fileIdRef>SampleInstallerActionsB</fileIdRef>
                    <parameterMaps>
                       <map>
                          <internalName>INSTALL_DIRECTORY_B</internalName>
                          <externalName>INSTALL_DIRECTORY_B</externalName>
                       </map>
                    </parameterMaps>
                 </installArtifact>
                 <uninstallArtifact>
                    <fileIdRef>SampleUnInstallerActionsB</fileIdRef>
                    <parameterMaps>
                       <map>
                          <internalName>INSTALL_DIRECTORY_B</internalName>
                          <externalName>INSTALL_DIRECTORY_B</externalName>
                       </map>
                    </parameterMaps>
                 </uninstallArtifact>
              </installArtifacts>
           </unit>
        </SIU>
     </installableUnit>
     </selectableContent>
```

```
<features>
   <feature featureID="Feature_A">
      <identity>
         <name>com.itso.Feature_A</name>
      </identity>
      <IUNameRef>SampleInstallerAppSIU_AMultiple</IUNameRef>
   </feature>

   <feature featureID="Feature_B">
      <identity>
         <name>com.itso.Feature_B</name>
      </identity>
      <IUNameRef>SampleInstallerAppSIU_BMultiple</IUNameRef>
   </feature>
</features>

<rootInfo>
   <schemaVersion>1.2.0</schemaVersion>
   <build>0</build>
</rootInfo>

<topology>
   <target id="tOperatingSystem" type="OSRT:Operating_System" >
      <selectionRequirements>
         <requirement name="OS">
            <alternative name="Windows_2000_Check_Alternative">
               <inlineCheck>
                  <property checkId="Windows_2000_Check">
                     <propertyName>OSType</propertyName>
                     <value>Windows 2000</value>
                  </property>
               </inlineCheck>
            </alternative>
            <alternative name="Windows_XP_Check_Alternative">
               <inlineCheck>
                  <property checkId="Windows_XP_Check">
                     <propertyName>OSType</propertyName>
                     <value>Windows XP</value>
                  </property>
               </inlineCheck>
            </alternative>
            <alternative name="Linux_Check-Alternative">
               <inlineCheck>
                  <property checkId="Linux_Check">
                     <propertyName>OSType</propertyName>
                     <pattern>LINUX.*</pattern>
                  </property>
               </inlineCheck>
```

```
                </alternative>
            </requirement>
        </selectionRequirements>
    </target>
</topology>

<files>
    <file id="SampleInstallerActionsA">
        <pathname>SampleInstallerActionsA.xml</pathname>
        <length>1392</length>
        <checksum />
    </file>
    <file id="SampleUnInstallerActionsA">
        <pathname>SampleUnInstallerActionsA.xml</pathname>
        <length>1346</length>
        <checksum />
    </file>
    <file id="testfile_A">
        <pathname>../FILES/SampleInstallerFile_A.txt</pathname>
        <length>79</length>
        <checksum />
    </file>
    <file id="SampleInstallerActionsB">
        <pathname>SampleInstallerActionsB.xml</pathname>
        <length>1392</length>
        <checksum />
    </file>
    <file id="SampleUnInstallerActionsB">
        <pathname>SampleUnInstallerActionsB.xml</pathname>
        <length>1346</length>
        <checksum />
    </file>
    <file id="testfile_B">
        <pathname>../FILES/SampleInstallerFile_B.txt</pathname>
        <length>79</length>
        <checksum />
    </file>
</files>
</iudd:rootIU>
```

Note the following changes in this deployment descriptor as compared to the single_packagedIU.xml in the previous example.

- The <installableUnit> is now contained within a <selectableContent> element that in turn is a sub element of <iudd:rootIU>. This was not the case for the single IU example. The <installableUnit> was a sub-element of <iudd:rootIU>. This change is attributed to the addition of the feature selection capability for this deployment descriptor.

- There is one more installable unit in this deployment descriptor. This IU is also a sub-element of the same <selectableContent> element. These IUs are eventually selected to be installed/uninstalled when a feature is selected.
- There are two sets of artifact files. One set is for the IU named SampleInstallerAppSIU_AMultiple and the other one for the IU named SampleInstallerAppSIU_BMultiple. Each set contains one install artifact file and one uninstall artifact file. The previous example had just one set of such files, as there was only one IU.
- There are different files to be copied by each IU's install operation. These files are referenced as testfile_A and testfile_B in the deployment descriptor and in the artifact files. As obvious from the names, the first file is used by the first IU and the second one is used by the second IU. In the previous example, there was a single IU and, hence, a single text file.
- A new element is introduced as compared to the deployment descriptor of the previous example. This element is <features> and is mandatory for feature selection. Its sub-elements are <feature> elements. Each such element represents one feature that can be selected by the user. The value of the featureID attribute must match the feature ID that is provided by the user and it is case-sensitive. A brief description of this element can be found in 3.1.11, "Feature element" on page 57.

The artifact files remain the same. Now, two more artifact files are introduced. Shown below are the sources for each artifact file.

This is the install artifact file for the first IU.

Example 6-14 SampleInstallerActionsA.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<action:artifact
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oca="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/act
ion action.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions osActions.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE base.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/command command.xsd">
   <artifactType>Install</artifactType>
   <artifactSchemaVersion>1.2.0</artifactSchemaVersion>
   <actionGroup xsi:type="oca:OsActionGroup">
   <actions>
      <addDirectory>
```

```
         <directory>
            <location>./</location>
            <name>$(INSTALL_DIRECTORY_A)</name>
         </directory>
      </addDirectory>

      <addFile>
         <file>
            <location>./$(INSTALL_DIRECTORY_A)/</location>
            <name>SampleInstallerFile_A.txt</name>
            <source>
               <fileIdRef>testfile_A</fileIdRef>
            </source>
         </file>
      </addFile>
   </actions>
   </actionGroup>
</action:artifact>
```

The second artifact file is shown next.

Example 6-15 SampleInstallerActionsB.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<action:artifact
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oca="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/act
ion action.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions osActions.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE base.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/command command.xsd">
   <artifactType>Install</artifactType>
   <artifactSchemaVersion>1.2.0</artifactSchemaVersion>
   <actionGroup xsi:type="oca:OsActionGroup">
   <actions>
      <addDirectory>
         <directory>
            <location>./</location>
            <name>$(INSTALL_DIRECTORY_B)</name>
         </directory>
      </addDirectory>

      <addFile>
         <file>
```

```
            <location>./$(INSTALL_DIRECTORY_B)/</location>
            <name>SampleInstallerFile_B.txt</name>
            <source>
               <fileIdRef>testfile_B</fileIdRef>
            </source>
         </file>
      </addFile>
   </actions>
   </actionGroup>
</action:artifact>
```

Next is the uninstall artifact file for the first IU.

Example 6-16 SampleUnInstallerActionsA.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<action:artifact
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oca="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/act
ion action.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions osActions.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE base.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/command command.xsd">
   <artifactType>Uninstall</artifactType>
   <artifactSchemaVersion>1.2.0</artifactSchemaVersion>
   <actionGroup xsi:type="oca:OsActionGroup">
      <actions>
         <removeFile>
            <location>./$(INSTALL_DIRECTORY_A)</location>
            <name>SampleInstallerFile_A.txt</name>
         </removeFile>
         <removeDirectory>
            <location>./</location>
            <name>$(INSTALL_DIRECTORY_A)</name>
         </removeDirectory>
      </actions>
   </actionGroup>
</action:artifact>
```

Below is the uninstall artifact for the second IU.

Example 6-17 SampleUnInstallerActionsB.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<action:artifact
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oca="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/act
ion action.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions osActions.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE base.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/command command.xsd">
   <artifactType>Uninstall</artifactType>
   <artifactSchemaVersion>1.2.0</artifactSchemaVersion>
   <actionGroup xsi:type="oca:OsActionGroup">
      <actions>
         <removeFile>
            <location>./$(INSTALL_DIRECTORY_B)</location>
            <name>SampleInstallerFile_B.txt</name>
         </removeFile>
         <removeDirectory>
               <location>./</location>
               <name>$(INSTALL_DIRECTORY_B)</name>
         </removeDirectory>
      </actions>
   </actionGroup>
</action:artifact>
```

6.5.2 Installation program

The installation program remains the same as in the previous example.

6.5.3 Payload

There are two text files. Each feature uses a single text file for install and uninstall operations.

The text file to be copied by the first feature is as follows.

Example 6-18 SampleInstallerFile_A.txt

```
This is a sample file for demonstrating SampleInstaller. This is for feature A.
```

The text file to be copied by the second feature is as follows.

Example 6-19 SampleInstallerFile_B.txt

```
This is a sample file for demonstrating SampleInstaller. This is for feature B.
```

6.5.4 Response file

The Response.properties has also changed from the previous example.

Example 6-20 Response.properties

```
Variable#SoftwareIUTypeID[AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA,1.0,RootIUTypeID[111
1222233334444555566667777888,1.1]]#INSTALL_DIRECTORY_A=CustomA
#Variable#SoftwareIUTypeID[BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB,1.0,RootIUTypeID[11
1122223333444455556666777788888,1.1]]#INSTALL_DIRECTORY_B=CustomB
Feature#RootIUTypeID[111122223333444455556666777788888,1.1]#com.itso.Feature_A
Feature#RootIUTypeID[111122223333444455556666777788888,1.1]#com.itso.Feature_B
```

The above Response.properties file contains two variable properties and two feature properties. Any of them can be excluded by adding a hash (#) character at the front of the line. Make sure that at least one feature property is uncommented, otherwise no installation would take place. Variable overriding is optional and both of the variables can be commented here.

6.5.5 Execution

The install program is run in the same way as done earlier, but with a few minor changes in the parameters. The names of the deployment descriptors and properties file have changed. Hence, the command would be:

```
java com.itso.ac.si.SampleInstaller Meta-INF/packagedIU.xml
Response.properties create
```

6.5.6 Output

The result of performing this installation is demonstrated in the examples below. Note that both the features were selected for installation and only INSTALL_DIRECTORY_A was overridden with the value Custom.

Example 6-21 Initial directory listing and listiu execution

```
C:\Sample2>dir
 Volume in drive C has no label.
 Volume Serial Number is 5CC0-9DEE

 Directory of C:\Sample2

07/07/2004  09:26 AM    <DIR>          .
```

```
07/07/2004  09:26 AM    <DIR>          ..
07/07/2004  09:17 AM    <DIR>          FILES
07/07/2004  09:17 AM    <DIR>          Meta-INF
07/06/2004  11:10 PM               440 Response.properties
               1 File(s)            440 bytes
               4 Dir(s)  48,883,556,352 bytes free

C:\Sample2>listiu
ACUIR00002I List is empty!
```

Next, we perform the installation, as shown below.

Example 6-22 Invoking the installation with the create operation

```
C:\Sample2>java com.itso.ac.si.SampleInstaller Meta-INF/packagedIU.xml
Response.properties create
****** Sample Installer ******
Simple program to demonstrate
some features of the
Solution Installation for
Autonomic Computing toolkit
******************************


Check name:Windows_2000_Check
 Check status: false

Check name:Windows_XP_Check
 Check status: true

Property name : OSType
Property value : Windows XP

Check name:Linux_Check
 Check status: false

Parameter Overrides:
 Variable : INSTALL_DIRECTORY_A
 Value : CustomA


FeatureId : com.itso.Feature_B
FeatureId : com.itso.Feature_A
Hosting Environment found is:
mrid:http://w3.ibm.com/namespaces/2003/OS_componen
tTypes:Operating_System

Begin Create Operation
```

```
SampleInstaller has recieved a changeOccurred Event: SUBMITTED
SampleInstaller has recieved a changeOccurred Event: EXECUTING
SampleInstaller has recieved a changeOccurred Event: EXECUTING
SampleInstaller has recieved a changeCompleted Event: COMPLETED
```

Now a listIU command and directory listings show that the installation has completed and directories for the two features have been created and populated. Note that one of the directories is the default directory name and the other is the custom name, based on having chosen only the one variable override in our Respponse.properties file.

Example 6-23 Results of installation

```
C:\Sample2>listiu
IU      UUID: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  Name:
SampleInstallerAppIU_AMult Version: 1.0
IU      UUID: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  Name:
SampleInstallerAppIU_BMult Version: 1.0
IU      RootIU UUID: 11112222333344445555666677778888   Name:
SampleInstallerRoot_Multiple      Version: 1.1
Feature (RootIU UUID: 11112222333344445555666677778888  Name:
com.itso.Feature_A)Feature Name:com.itso.Feature_A
Feature (RootIU UUID: 11112222333344445555666677778888  Name:
com.itso.Feature_B)Feature Name:com.itso.Feature_B

C:\Sample2>dir
 Volume in drive C has no label.
 Volume Serial Number is 5CC0-9DEE

 Directory of C:\Sample2

07/07/2004  09:28 AM    <DIR>          .
07/07/2004  09:28 AM    <DIR>          ..
07/07/2004  09:28 AM    <DIR>          CustomA
07/07/2004  09:28 AM    <DIR>          defaultB
07/07/2004  09:17 AM    <DIR>          FILES
07/07/2004  09:17 AM    <DIR>          Meta-INF
07/06/2004  11:10 PM               440 Response.properties
               1 File(s)            440 bytes
               6 Dir(s)  48,883,482,624 bytes free

C:\Sample2>dir CustomA
 Volume in drive C has no label.
 Volume Serial Number is 5CC0-9DEE

 Directory of C:\Sample2\CustomA

07/07/2004  09:28 AM    <DIR>          .
```

```
07/07/2004  09:28 AM    <DIR>          ..
07/07/2004  09:28 AM                79 SampleInstallerFile_A.txt
               1 File(s)             79 bytes
               2 Dir(s)  48,883,482,624 bytes free

C:\Sample2>dir defaultB
 Volume in drive C has no label.
 Volume Serial Number is 5CC0-9DEE

 Directory of C:\Sample2\defaultB

07/07/2004  09:28 AM    <DIR>          .
07/07/2004  09:28 AM    <DIR>          ..
07/07/2004  09:28 AM                79 SampleInstallerFile_B.txt
               1 File(s)             79 bytes
               2 Dir(s)  48,883,482,624 bytes free
```

6.6 Summary

This chapter described a very simple installation program that utilizes the Solution Installation toolkit. Aside from the structure of the installation program, we have provided two examples of using it with deployment descriptors, artifacts, and properties files that demonstrate the use of variables and feature selection.

The installation program shown here is not designed to be robust or to be used in a production environment, but rather demonstrates some of the basic concepts and techniques that might be used in a more robust implementation.

7

Creating and finalizing the installation package

In this chapter, we describe the different components required for a complete installation package and various ways of bundling the package.

Note: The examples and descriptions in this chapter are based on our work with an early version of the Solution Installation for Autonomic Computing toolkit. Therefore, some differences may exist from the generally available version.

7.1 Components of a package

Once the deployment descriptor, media descriptor, action artifacts file(s), and payload are ready, the next step is to bundle all the various files together and make a complete package. A complete installable package can contain the following:

- Deployment descriptor

  Deployment descriptor contains the logical structure of the package. It defines the checks, dependencies, files and other important information. For more details on this, please refer to Chapter 3, “Descriptors and artifacts” on page 33.

- Media descriptor

  Media descriptor defines the physical location of the files that are referenced in the deployment descriptor. This is an optional component and if it is absent, the default path is taken for the files. The default path is the directory where the deployment descriptor is located. For more details on this, please refer to Chapter 3, “Descriptors and artifacts” on page 33.

- Artifact files

  As part of the install process, actions are executed in the hosting environments. These actions are sometimes referred to as artifacts, but are described either in a deployment descriptor or in an action artifacts file. If a separate action artifacts file is used, a reference to the artifacts file is provided in the deployment descriptor. This is also an optional file. For more details on this, please refer to 3.3.1, “Action artifacts” on page 65.

- Custom installation code

  This is the installation or change management program that is used for the installation. This code can be provided externally or can be a part of the IU package.

- Payload files

  Payload is a term used for the files to be installed in the hosting environment. These files can be executables, binaries, libraries, and so on. Payload files are copied to the hosting environment. The payload may also include non-functional files such as read me files, installation instructions, and other informational files. This is a mandatory component.

- Optional files

  There may be more files that are neither descriptors nor payload files. These temporary files may be used by the installation program during installation. Examples of such are JARs, JVMs, and so on. They help to make the package self-contained, without any additional dependency on the

environment. They are specified neither in the deployment descriptor nor in the media descriptor. They are included in the IU package to make the package self-contained, so that it can work without any additional prerequisites.

Note: Deployment descriptor, media descriptor, and artifact files are collectively known as manifest files.

7.2 Conventions to arranging files in a package

This section describes the recommended structure for bundling a software package. The files can be arranged in any hierarchy within the package. This structure is totally up to the creator of the package. Even though the structure of the package is totally up to the creator, in the case of using removable media (such as CDs), all of the manifest files must be placed on the first CD. Non-manifest files can be placed anywhere and bound by the media descriptor.

Relative references to files are resolved with respect to the location of the deployment descriptor. If a media descriptor is present, the location would be calculated using the path given in the media descriptor and the relative path of the file as defined in the deployment descriptor.

Solution Installation does not provide any means of bundling a package. The bundling of a software package is up to the creator and installation program that is to be used.

A software package is defined by the directories, files and file formats that are included within it. These objects must exist in a specific structure and must follow package-specific naming guidelines.

7.2.1 root(/) directory

The root directory should contain the files and sub-directories for the following types of files:

- Installation engine files
- Files required by the installation program to implement processing logic that is specific to the package

These types of files are not described within the descriptor files. The directory and file structure within the root directory is dependent on the installation program. Examples of contents within the root directory are class files for the installation program, properties files required by the installation program, and so on.

7.2.2 META-INF sub-directory

The manifest files should be placed under ./META-INF. This folder is meant to contain the files required by the installer (like the deployment descriptor, artifact files, and so on).

7.2.3 FILES sub-directory

The payload files should be placed under ./FILES. The files can be arranged according to the package structure within ./FILES. These are the files that are described within the <files> element in the deployment descriptor and are required during the installation of the package.

7.3 Types of package bundles

There are many types of standard package types used across the industry. We will discuss four of the most commonly used types:

- Fixed-size removable media
- Single zip file
- Self-extractable single executable
- Network download

7.3.1 Fixed-size removable media

This type of packaging format is very common. Some of the media types used for such format are diskettes and CDs. The package is split into similar sized groupings and are stored on one of the media. Each diskette/CD is labelled as a volume. Information about the volume that will contain a specific file is provided in the media descriptor. This type of format is generally used in those cases where an installable package is very large in size and cannot fit onto a single media.

Manifest files need to be present under ./META-INF. If using removable media, the ./META-INF directory must be on the first CD or other removable media. An installation program should always look for the manifest files under ./META-INF. The rest of the files can be anywhere on any of the media. File locations are specified in the media descriptor using the volume number and path within that volume.

As mentioned above, the manifest files are located on the first volume. If the media descriptor is not present, the current path of the deployment descriptor is considered as the root path for prepending to relative paths given in the deployment descriptor. The location of each file is specified in the media

descriptor. Information such as which volume will contain which file is defined in the media descriptor. Hence, the media descriptor becomes mandatory in such formats.

Example 7-1 is a sample media descriptor of a package in fixed-size format. file1 is referenced in the deployment descriptor. Suppose that the relative path of file1 is bin/abc.bat. Hence, the absolute path would be /Files/bin/abc.bat on CD#4.

Example 7-1 Media descriptor extract for removable media

```
<fileSource>
   <fileIdRef>file1</fileIdRef>
   <location>
      <physicalLocation>
         <removableMedia volumeNumber="4" type="CDROM">
            <path>/FILES</path>
         </removableMedia>
      </physicalLocation>
   </location>
</fileSource>
```

7.3.2 Single zip file

In this case, all of the installable unit (IU) files are packaged in a single zip file. This type of installation format is often used when the package is downloadable from the Web. In these situations, all the manifest files, payload files and any other optional files are bundled in a zip file. The zip file maintains a logical directory structure. The directories referenced below would be within the logical directory structure maintained within the zip file.

Manifest files need to be present under ./META-INF. An installation program should always look for the manifest files under ./META-INF. The rest of the files can be anywhere inside or outside the zip. But we recommend that you have all files in a single zip. The file locations are specified in the media descriptor.

Example 7-2 is a sample media descriptor of a package in single zip format. file2 is referenced in the deployment descriptor. Suppose that the relative path of file2 is bin/def.bat. Hence, the absolute path would be ./Files/bin/def.bat in the MyPkg.zip file.

Example 7-2 Zip file media descriptor

```
<fileSource>
   <fileIdRef>file2</fileIdRef>
   <location>
      <physicalLocation>
         <ZipFile>
```

```
            <networkSource>./MyPkg.zip</networkSource>
            <path>FILES</path>
         </ZipFile>
      </physicalLocation>
   </location>
</fileSource>
```

Modifications to SampleInstaller code

Suppose that you have a ZIP/JAR file that contains the manifest files. If you have to install the package, you may extract the zip and install it using the sample installation program created in Chapter 6, “Creating an installation program” on page 93. But there is an alternative to this. Solution Installation provides APIs to read the manifest files from a zip file. The following code sample shows an addition to the SampleInstaller. In the original sample the following line of code was used to create the MediaInfo object using the deployment descriptor URI (deployDescriptor).

Example 7-3 Code before modification

```
mi = new MediaInfo(deployDescriptor, null, null);
```

Since we are now passing a ZIP file to the installation program instead of a deployment descriptor, we cannot instantiate a MediaInfo object using the deployment descriptor URI. Hence, first we will check if the deployDescriptor is a valid package file name or not. A valid package file name must end with .zip or .jar. If its a valid package name, then we would instantiate a MediaInfo object.

Example 7-4 Code after modification

```
   if(MediaReaderUtil.isValidPackageFileName(deployDescriptor))
      mi = new MediaInfo(null, null, deployDescriptor);
   else if (MediaReaderUtil.isValidIUDDFileName(deployDescriptor))
      mi = new MediaInfo(deployDescriptor, null, null);
```

The MediaInfo constructor has three arguments. The definition of the constructor is as follows:

```
public MediaInfo(String iuddFileName, String iumdFileName, String
iuPkgFileName)
```

Where:

- iuddFileName is the URI of a deployment descriptor.
- iumdFileName is the URI of a media descriptor.
- iuPkgFileName is the URI of a ZIP/JAR bundle.

At least one of the arguments must be non-null.

By now it should be clear that if the deployDescriptor is a valid package name, we pass it on as third argument to MediaInfo constructor. If the deployDescriptor is a valid deployment descriptor, we pass it on as the first argument to the MediaInfo constructor.

The rest of the code remains the same, if the new installation program is SampleInstallerPkg.java.

This program will be run as follows:

```
java com.itso.ac.si.SampleInstallerPkg sample.jar Response.properties
install
```

sample.jar should at least contain ./META-INF, as in previous examples. The ./FILES directory is optional and it depends on the deployment descriptor. But sample.jar must contain the payload files. How these payload files are structured and arranged can be described in the deployment descriptor. That means that the manifest files must be under ./META-INF and the payload files can be anywhere in the sample.jar and clearly defined in the deployment descriptor.

This new installation program can also be run with a deployment descriptor URI passed as an argument. The below command should look familiar.

```
java com.itso.ac.si.SampleInstallerPkg META-INF\packagedIU.xml
Response.properties install
```

7.3.3 Self-extractable single executable

These are self-contained executable files. They would contain all the required files for installation. Usually, they are in platform-specific binaries that when launched can unpack as necessary and launch a program packaged within the bundle. Certainly, this technology can be used to package all of the required manifest and payload files along with an installation program that would install a software package.

Once a package has been constructed using one of the techniques we have already covered, the entire set of files could be bundled together into a self-extracting file for simpler distribution and deployment.

7.3.4 Network download

In such formats, the required files are located on a network. These files can be accessed using a URI. As usual, the media descriptor would contain the information about the URIs of the files to be downloaded from the network.

Example 6.5 is a sample media descriptor of a package in network download format. Let us assume that file3 is referenced in the deployment descriptor with a

relative path of ./Files/file3.txt. The absolute location of this file as defined by the example below would be http://somelocation.com/Files/files3.txt.

Example 7-5 Media descriptor for network download

```
<fileSource>
   <fileIdRef>file3</fileIdRef>
   <location>
      <physicalLocation>
         <URL>http://somelocation.com</URL>
      </physicalLocation>
   </location>
</fileSource>
```

7.3.5 Combination of formats

There is no constraint that a single package should always use only one of the formats described above. A package can use combinations of different formats together. For example, consider a scenario where the package should contain some basic files in the ZIP format. Depending on the requirements of the user or some other condition, optional files can be downloaded from a network location. In this case, the package would contain sufficient files for installation, but it would provide the flexibility to access and install additional features on demand.

7.4 Summary

This chapter has provided an overview of some of the options for packaging the manifest and payload files required for the installation of a software package. Standard packaging techniques can be used, such as removable media, zip files, and accessing files from a network location. The media descriptor provides the information as to the actual location of the files needed by the deployment descriptor.

8

Step-by-step example

In this chapter we provide a step-by-step example of using the Solution Installation toolkit and technologies to perform a software installation. We build on the information and sample installation program presented earlier in this publication.

Note: The examples and descriptions in this chapter are based on our work with an early version of the Solution Installation for Autonomic Computing toolkit. Therefore, some differences may exist from the generally available version.

8.1 Installation objectives

Let us first describe our objectives for this installation exercise. Our example will be quite simple compared to what you might need to accomplish in a production environment, but it will cover the basics and give you a sample to build upon.

In general, a typical software package has several components, some of which are listed below:

- Set of executables/binaries
- Set of libraries/dlls
- Set of script files
- Set of property/resource files
- Documentation files
- Set of samples
- Set of features to be selected during installation (like docs, samples, and so on)
- Batch script to set up the environment before using the application

Our package will also have a similar structure. Our sample package, keeping in line with its trivial nature, will be called ABC. The package has features, some of which are mandatory and others optional. For each feature, there are associated payload files that need to be written to the hosting environment during the installation process. ABC has a set of prerequisites that are to be fulfilled before the installation can take place. Details of each are given in following sections.

8.1.1 Prerequisites of the package

The prerequisites for the ABC package are as follows:

- Windows or Linux operating system
- 100 MB of free disk space

8.1.2 Folder structure

ABC requires a typical directory structure as described below:

bin	Contains executables/binaries, script files and dlls
lib	Contains libraries (JARs)
properties	Contains property/resource files
docs	Contains the documentation files

samples Contains samples

All these folders are created under INSTALL_BASE, where INSTALL_BASE is the location where ABC will be installed.

8.1.3 Features

ABC has the following features that are selectable by the user.

ABC Runtime This is a mandatory feature and selected by default. This feature creates folders bin, lib and properties. Also, it copies bin/abc.exe, bin/script1.bat. bin/script2.bat, lib/abc.jar, /lib/xyz.jar, properties/abc.properties and properties/def.properties to their respective folders.

Documentation This is an optional feature. Selection of this feature would create the docs folder and copy docs/index.html and docs/com/itso/ac/si/abc.html into it.

Samples This is an optional feature. Selection of this feature would create a samples folder and copy samples/sample1.exe to it.

Developer's kit This is an optional feature. Selection of this feature would copy lib/abc_dk.jar to lib.

8.1.4 Payload files

Based on the above description, our simple software package includes the following payload files.

- /bin
  - script1.bat
  - script2.bat
  - abc.exe
- /lib
  - abc.jar
  - xyz.jar
  - abc_dk.jar
- /properties
  - abc.properties
  - def.properties
- /docs
  - index.html

  - com\itso\ac\si\abc.html
- /samples
  - sample1.exe

8.2 Creating the descriptors and artifacts

After describing the logical structure of the package, let us create the manifest files.

8.2.1 Deployment descriptor

Our deployment descriptor will need to contain the following sections:

- Identity of the package

  This is the section that contains the name, UUID, and version of the package.
- List of variables

  This section contains variables used in the manifest files of the package.
- List of installable units

  This lists the mandatory installable units of the package. These IUs are not selectable by the user.
- List of optional installable units

  These are the IUs that are selected by the user and are optional.
- List of features

  All of the optional IUs are associated with a feature ID. All the feature IDs used in our deployment descriptor are listed here.
- List of probable hosting environment

  The package is targeted for a few specific hosting environments that are defined here.
- List of checks and requirements

  The checks and requirements to ensure that our target hosting environment meets the package prerequisites.
- List of files used in the package

  Payload files are mentioned here and reference to these files is used in the other manifest files.

Let us start with the IU definitions. Since there is one mandatory feature (runtime) and three optional features (documentation, samples and developer's kit), all of

these features are independent IUs. Documentation, samples, and the developer' kit are part of <selectableContent>.

The following shows the first part of our deployment descriptor (packagedIU.xml). It includes the necessary header information for the schema files and the rootIU identity information. Note that the UUIDs we used are obviously fabricated. we chose to do this so that references to the various UUIDs will be more easily accomplished throughout our example and descriptions.

Example 8-1 Deployment descriptor (packagedIU.xml) part 1

```
<?xml version="1.0" encoding="UTF-8"?>
<iudd:rootIU
xmlns:OSRT="http://www.ibm.com/namespaces/autonomic/OS_componentTypes"
xmlns:J2EERT="http://www.ibm.com/namespaces/autonomic/J2EE_RT"
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:rtype="http://www.ibm.com/namespaces/autonomic/solutioninstall/ResourceTy
pes"
xmlns:siu="http://www.ibm.com/namespaces/autonomic/solutioninstall/SIU"
xmlns:vsn="http://www.ibm.com/namespaces/autonomic/common/version"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUD
D iudd.xsd"
IUName="ABC_IU" targetRef="tOperatingSystem" >
   <identity>
      <name>ABC_Sample_Package</name>
      <UUID>12341234123412341234123412341234</UUID>
      <version>1.1</version>
   </identity>
   <variables>
      <variable name="INSTALL_BASE">
         <parameter defaultValue="/myapps/ABC_APP"/>
      </variable>
   </variables>
```

The next portion of our deployment descriptor (shown below), provides the definition for the mandatory installation unit (run time). It has its own identity section, ABC_Runtime_SIU. Next we define some variables that we will use in our action artifacts and our properties file. These variables allow a user to modify the location and name of certain installation directories and files.

After the variables section, we define various checks that will be used to validate our target environment. Notice that the checks defined here are all related to the operating system hosting environment, as that is the target of the installable unit. In our topology section, seen in Example 8-4 on page 160, we also define the

disk space check. That check is defined on the file system and not within the operating system hosting environment.

After the checks, we define and organize the requirements. As you can see, we have an operating system requirement that allows us to pass the requirement check if we have a valid version of Windows or Linux. The second requirement is for the disk space requirement that applies to the file system.

> **Tip:** Every <requirement> element represents one unique requirement. Suppose there are three such elements. Each of the elements should come true. There may be various <alternative> elements within a <requirement> element. If at least one of the alternatives comes true, the requirement is true.
>
> Also, with each <checkItem> in an alternative, you can have a "testValue" attribute where you can look for a false value instead of true.

The last section of this portion of the deployment descriptor defines the install and uninstall artifacts that will be used for the runtime IU. Parameter maps are included here to ensure that the variables defined earlier are usable within the artifacts. The fileIdRefs you see are referencing file definitions found later in the descriptor.

Example 8-2 Deployment descriptor (packagedIU.xml) part 2

```
<installableUnit targetRef="tOperatingSystem">
   <SIU IUName="ABC_Runtime" hostingEnvType="OSRT:Operating_System">
      <identity>
         <name>ABC_Runtime_SIU</name>
         <UUID>11111111111111111111111111111111</UUID>
         <version>1.0</version>
      </identity>
      <variables>
         <variable name="EXEC_DIR">
            <parameter defaultValue="bin"/>
         </variable>
         <variable name="PROPERTIES_DIR">
            <parameter defaultValue="props"/>
         </variable>
         <variable name="LIB_DIR">
            <parameter defaultValue="lib"/>
         </variable>
         <variable name="ENV_BAT">
            <parameter defaultValue="env.bat"/>
         </variable>
      </variables>
      <checks>
         <property checkId="Windows_Type_Check">
```

```
      <propertyName>OSType</propertyName>
      <pattern>Windows.*</pattern>
   </property>
   <version checkId="Windows_Version_Check">
      <propertyName>version</propertyName>
      <minVersion>4.0</minVersion>
      <maxVersion>5.5</maxVersion>
   </version>
   <property checkId="Linux_Check">
      <propertyName>OSType</propertyName>
      <pattern>LINUX</pattern>
   </property>
   <version checkId="Linux_Version_Check">
      <propertyName>version</propertyName>
      <minVersion>2.4</minVersion>
      <maxVersion>2.6</maxVersion>
   </version>
</checks>
<requirements>
   <requirement name="PlatformRequirements">
      <alternative name="Windows" priority="0">
         <checkItem checkIdRef="Windows_Type_Check"/>
         <checkItem checkIdRef="Windows_Version_Check"/>
      </alternative>
      <alternative name="Linux" priority="1">
         <checkItem checkIdRef="Linux_Check"/>
         <checkItem checkIdRef="Linux_Version_Check"/>
      </alternative>
   </requirement>
   <requirement name="CapacityRequirements">
      <alternative name="CapacityAlternatives" priority="0">
         <checkItem checkIdRef="Diskspace_Check"/>
      </alternative>
   </requirement>
</requirements>
<unit>
   <installArtifacts>
      <installArtifact>
         <fileIdRef>InstallerActionsRuntime</fileIdRef>
         <parameterMaps>
            <map>
               <internalName>INSTALL_BASE</internalName>
               <externalName>INSTALL_BASE</externalName>
            </map>
            <map>
               <internalName>EXEC_DIR</internalName>
               <externalName>EXEC_DIR</externalName>
            </map>
            <map>
```

```
                        <internalName>PROPERTIES_DIR</internalName>
                        <externalName>PROPERTIES_DIR</externalName>
                     </map>
                     <map>
                        <internalName>LIB_DIR</internalName>
                        <externalName>LIB_DIR</externalName>
                     </map>
                     <map>
                        <internalName>ENV_BAT</internalName>
                        <externalName>ENV_BAT</externalName>
                     </map>
                  </parameterMaps>
               </installArtifact>
               <uninstallArtifact>
                  <fileIdRef>UninstallerActionsRuntime</fileIdRef>
                  <parameterMaps>
                     <map>
                        <internalName>INSTALL_BASE</internalName>
                        <externalName>INSTALL_BASE</externalName>
                     </map>
                     <map>
                        <internalName>EXEC_DIR</internalName>
                        <externalName>EXEC_DIR</externalName>
                     </map>
                     <map>
                        <internalName>PROPERTIES_DIR</internalName>
                        <externalName>PROPERTIES_DIR</externalName>
                     </map>
                     <map>
                        <internalName>LIB_DIR</internalName>
                        <externalName>LIB_DIR</externalName>
                     </map>
                     <map>
                        <internalName>ENV_BAT</internalName>
                        <externalName>ENV_BAT</externalName>
                     </map>
                  </parameterMaps>
               </uninstallArtifact>
            </installArtifacts>
         </unit>
      </SIU>
   </installableUnit>
```

The next section of the deployment descriptor includes the selectable content. This includes the definitions of the optional installable units (documentation, samples and developer's kit). There is nothing special to note here, other than the fact that you will notice that additional variables that are specific to the optional installable unit can be defined here. Variables, such as INSTALL_BASE,

that were defined for the mandatory installable unit can also be used in conjunction with these units.

Example 8-3 Deployment descriptor (packagedIU.xml) part 3

```
   <selectableContent>
      <installableUnit targetRef="tOperatingSystem">
         <SIU IUName="ABC_Documentation"
hostingEnvType="OSRT:Operating_System">
            <identity>
               <name>ABC_Documentation_SIU</name>
               <UUID>22222222222222222222222222222222</UUID>
               <version>1.0</version>
            </identity>
            <variables>
               <variable name="DOC_DIR">
                  <parameter defaultValue="doc"/>
               </variable>
            </variables>
            <unit>
               <installArtifacts>
                  <installArtifact>
                     <fileIdRef>InstallerActionsDocs</fileIdRef>
                     <parameterMaps>
                        <map>
                           <internalName>INSTALL_BASE</internalName>
                           <externalName>INSTALL_BASE</externalName>
                        </map>
                        <map>
                           <internalName>DOC_DIR</internalName>
                           <externalName>DOC_DIR</externalName>
                        </map>
                     </parameterMaps>
                  </installArtifact>
                  <uninstallArtifact>
                     <fileIdRef>UninstallerActionsDocs</fileIdRef>
                     <parameterMaps>
                        <map>
                           <internalName>INSTALL_BASE</internalName>
                           <externalName>INSTALL_BASE</externalName>
                        </map>
                        <map>
                           <internalName>DOC_DIR</internalName>
                           <externalName>DOC_DIR</externalName>
                        </map>
                     </parameterMaps>
                  </uninstallArtifact>
               </installArtifacts>
            </unit>
```

```
   </SIU>
</installableUnit>
<installableUnit targetRef="tOperatingSystem">
   <SIU IUName="ABC_Samples" hostingEnvType="OSRT:Operating_System">
      <identity>
         <name>ABC_Samples_SIU</name>
         <UUID>3333333333333333333333333333333</UUID>
         <version>1.0</version>
      </identity>
      <variables>
         <variable name="SAMPLES_DIR">
            <parameter defaultValue="samples"/>
         </variable>
      </variables>
      <unit>
         <installArtifacts>
            <installArtifact>
               <fileIdRef>InstallerActionsSamples</fileIdRef>
               <parameterMaps>
                  <map>
                     <internalName>INSTALL_BASE</internalName>
                     <externalName>INSTALL_BASE</externalName>
                  </map>
                  <map>
                     <internalName>SAMPLES_DIR</internalName>
                     <externalName>SAMPLES_DIR</externalName>
                  </map>
               </parameterMaps>
            </installArtifact>
            <uninstallArtifact>
               <fileIdRef>UninstallerActionsSamples</fileIdRef>
               <parameterMaps>
                  <map>
                     <internalName>INSTALL_BASE</internalName>
                     <externalName>INSTALL_BASE</externalName>
                  </map>
                  <map>
                     <internalName>SAMPLES_DIR</internalName>
                     <externalName>SAMPLES_DIR</externalName>
                  </map>
               </parameterMaps>
            </uninstallArtifact>
         </installArtifacts>
      </unit>
   </SIU>
</installableUnit>
<installableUnit targetRef="tOperatingSystem">
   <SIU IUName="ABC_DK" hostingEnvType="OSRT:Operating_System">
      <identity>
```

```
   <name>ABC_DK_SIU</name>
   <UUID>44444444444444444444444444444444</UUID>
   <version>1.0</version>
</identity>
<variables>
   <variable name="ABC_DK_JAR1">
      <parameter defaultValue="abc_dk.jar"/>
   </variable>
   <variable name="DK_LIB_DIR">
      <parameter defaultValue="lib"/>
   </variable>
</variables>
<unit>
   <installArtifacts>
      <installArtifact>
         <fileIdRef>InstallerActionsDK</fileIdRef>
         <parameterMaps>
            <map>
               <internalName>INSTALL_BASE</internalName>
               <externalName>INSTALL_BASE</externalName>
            </map>
            <map>
               <internalName>DK_LIB_DIR</internalName>
               <externalName>DK_LIB_DIR</externalName>
            </map>
            <map>
               <internalName>ABC_DK_JAR1</internalName>
               <externalName>ABC_DK_JAR1</externalName>
            </map>
         </parameterMaps>
      </installArtifact>
      <uninstallArtifact>
         <fileIdRef>UninstallerActionsDK</fileIdRef>
         <parameterMaps>
            <map>
               <internalName>INSTALL_BASE</internalName>
               <externalName>INSTALL_BASE</externalName>
            </map>
            <map>
               <internalName>ABC_DK_JAR1</internalName>
               <externalName>ABC_DK_JAR1</externalName>
            </map>
            <map>
               <internalName>DK_LIB_DIR</internalName>
               <externalName>DK_LIB_DIR</externalName>
            </map>
         </parameterMaps>
      </uninstallArtifact>
   </installArtifacts>
```

```
            </unit>
         </SIU>
      </installableUnit>
   </selectableContent>
```

In the above example, it should be noted that the internal_name and external_name values within the map elements do not have to match. The external name would be used in any artifacts associated with the deployment descriptor.

The next section of the deployment descriptor includes the definition of the features, root information and topology. Note that in the topology section is where we define the file system target. This is purely so that we can perform the check (also shown in the definition) for available space. This check is used in the requirements checking performed in Example 8-2 on page 154.

Example 8-4 Deployment descriptor (packagedIU.xml) part 4

```
   <features>
      <feature featureID="Feature_Samples">
         <identity>
            <name>Feature_Samples</name>
         </identity>
         <IUNameRef>ABC_Samples</IUNameRef>
      </feature>
      <feature featureID="Feature_Documentation">
         <identity>
            <name>Feature_Documentation</name>
         </identity>
         <IUNameRef>ABC_Documentation</IUNameRef>
      </feature>
      <feature featureID="Feature_DK">
         <identity>
            <name>Feature_DK</name>
         </identity>
         <IUNameRef>ABC_DK</IUNameRef>
      </feature>
   </features>
   <rootInfo>
      <schemaVersion>1.2.0</schemaVersion>
      <build>0</build>
   </rootInfo>
   <topology>
      <target id="fs" type="FileSystem">
         <checks>
            <capacity checkId="Diskspace_Check" targetRef="fs">
               <propertyName>availableSpace</propertyName>
               <value>100000</value>
```

```
            </capacity>
         </checks>
      </target>
      <target id="tOperatingSystem" type="OSRT:Operating_System"/>
   </topology>
```

The final portion of the deployment descriptor includes the definition of all the files that will be used. These include other artifact files such as the action descriptors, as well as the payload files.

Example 8-5 Deployment descriptor (packagedIU.xml) part 5

```
   <files>
      <file id="InstallerActionsRuntime">
         <pathname>InstallerActionsRuntime.xml</pathname>
         <length>3747</length>
         <checksum/>
      </file>
      <file id="UninstallerActionsRuntime">
         <pathname>UninstallerActionsRuntime.xml</pathname>
         <length>2517</length>
         <checksum/>
      </file>
      <file id="InstallerActionsDocs">
         <pathname>InstallerActionsDocs.xml</pathname>
         <length>1744</length>
         <checksum/>
      </file>
      <file id="UninstallerActionsDocs">
         <pathname>UninstallerActionsDocs.xml</pathname>
         <length>1411</length>
         <checksum/>
      </file>
      <file id="InstallerActionsSamples">
         <pathname>InstallerActionsSamples.xml</pathname>
         <length>1387</length>
         <checksum/>
      </file>
      <file id="UninstallerActionsSamples">
         <pathname>UninstallerActionsSamples.xml</pathname>
         <length>128</length>
         <checksum/>
      </file>
      <file id="InstallerActionsDK">
         <pathname>InstallerActionsDK.xml</pathname>
         <length>1550</length>
         <checksum/>
      </file>
      <file id="UninstallerActionsDK">
```

```
   <pathname>UninstallerActionsDK.xml</pathname>
   <length>1151</length>
   <checksum/>
</file>
<file id="env_bat">
   <pathname>../FILES/env.bat</pathname>
   <length>113</length>
   <checksum/>
</file>
<file id="abc_exe">
   <pathname>../FILES/bin/abc.exe</pathname>
   <length>4460</length>
   <checksum/>
</file>
<file id="script1">
   <pathname>../FILES/bin/script1.bat</pathname>
   <length>111</length>
   <checksum/>
</file>
<file id="script2">
   <pathname>../FILES/bin/script2.bat</pathname>
   <length>137</length>
   <checksum/>
</file>
<file id="index">
   <pathname>../FILES/docs/index.html</pathname>
   <length>197</length>
   <checksum/>
</file>
<file id="abc_html">
   <pathname>../FILES/docs/com/itso/ac/si/abc.html</pathname>
   <length>85</length>
   <checksum/>
</file>
<file id="abc_jar">
   <pathname>../FILES/lib/abc.jar</pathname>
   <length>2154</length>
   <checksum/>
</file>
<file id="xyz_jar">
   <pathname>../FILES/lib/xyz.jar</pathname>
   <length>2169</length>
   <checksum/>
</file>
<file id="abc_dk_jar">
   <pathname>../FILES/lib/abc_dk.jar</pathname>
   <length>2175</length>
   <checksum/>
</file>
```

```
      <file id="abc_props">
         <pathname>../FILES/properties/abc.properties</pathname>
         <length>61</length>
         <checksum/>
      </file>
      <file id="def_props">
         <pathname>../FILES/properties/def.properties</pathname>
         <length>58</length>
         <checksum/>
      </file>
      <file id="samples_exe">
         <pathname>../FILES/samples/sample1.exe</pathname>
         <length>4495</length>
         <checksum/>
      </file>
   </files>
</iudd:rootIU>
```

8.2.2 Artifact files

There are separate artifact files for the create and delete operations of each feature (including the runtime feature). These files are named as follows:

- InstallerActionsDK.xml
- InstallerActionsDocs.xml
- InstallerActionsRuntime.xml
- InstallerActionsSamples.xml
- UninstallerActionsDK.xml
- UninstallerActionsDocs.xml
- UninstallerActionsRuntime.xml
- UninstallerActionsSamples.xml

All of these files are placed under /META-INF.

These files and brief descriptions are provided below.

ABC Runtime artifacts

The following figures show the bodies of the install and uninstall artifacts for the ABC run-time installable unit.

As can be seen below, the actions for installation include the creation of the directory structure (using variables such as INSTALL_BASE) defined in the deployment descriptor, add files to the directory structure, and in one case updating the contents of the file to reflect the path specified in a variable.

Example 8-6 Runtime install artifact

```
<?xml version="1.0" encoding="UTF-8"?>
<action:artifact
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oca="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/act
ion action.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions osActions.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE base.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/command command.xsd">
   <artifactType>Install</artifactType>
   <artifactSchemaVersion>1.2.0</artifactSchemaVersion>
   <actionGroup xsi:type="oca:OsActionGroup">
      <actions>
         <addDirectory>
            <directory>
               <location>/</location>
               <name>$(INSTALL_BASE)</name>
            </directory>
         </addDirectory>
         <addDirectory>
            <directory>
               <location>$(INSTALL_BASE)</location>
               <name>$(EXEC_DIR)</name>
            </directory>
         </addDirectory>
         <addDirectory>
            <directory>
               <location>$(INSTALL_BASE)</location>
               <name>$(LIB_DIR)</name>
            </directory>
         </addDirectory>
         <addDirectory>
            <directory>
               <location>$(INSTALL_BASE)</location>
               <name>$(PROPERTIES_DIR)</name>
            </directory>
         </addDirectory>
         <addFile>
            <file>
               <location>$(INSTALL_BASE)</location>
               <name>$(ENV_BAT)</name>
               <source>
                  <fileIdRef>env_bat</fileIdRef>
               </source>
            </file>
```

```
          </addFile>
          <updateTextFile>
             <pathname>$(INSTALL_BASE)/$(ENV_BAT)</pathname>
             <!--The following action replaces all the occurrences of the
string toBeReplaced with the string REPLACED. It is possible to use a generic
regular expression to specify the search pattern.-->
             <replaceString>
                <text>$(INSTALL_BASE)</text>
                <occurrence>all</occurrence>
                <searchPattern>INSTALL_BASE</searchPattern>
             </replaceString>
          </updateTextFile>
          <addFile>
             <file>
                <location>$(INSTALL_BASE)/$(EXEC_DIR)</location>
                <name>abc.exe</name>
                <source>
                   <fileIdRef>abc_exe</fileIdRef>
                </source>
             </file>
          </addFile>
          <addFile>
             <file>
                <location>$(INSTALL_BASE)/$(EXEC_DIR)</location>
                <name>script1.bat</name>
                <source>
                   <fileIdRef>script1</fileIdRef>
                </source>
             </file>
          </addFile>
          <addFile>
             <file>
                <location>$(INSTALL_BASE)/$(EXEC_DIR)</location>
                <name>script2.bat</name>
                <source>
                   <fileIdRef>script2</fileIdRef>
                </source>
             </file>
          </addFile>
          <addFile>
             <file>
                <location>$(INSTALL_BASE)/$(LIB_DIR)</location>
                <name>abc.jar</name>
                <source>
                   <fileIdRef>abc_jar</fileIdRef>
                </source>
             </file>
          </addFile>
          <addFile>
```

```
            <file>
               <location>$(INSTALL_BASE)/$(LIB_DIR)</location>
               <name>xyz.jar</name>
               <source>
                  <fileIdRef>xyz_jar</fileIdRef>
               </source>
            </file>
         </addFile>
         <addFile>
            <file>
               <location>$(INSTALL_BASE)/$(PROPERTIES_DIR)</location>
               <name>abc.properties</name>
               <source>
                  <fileIdRef>abc_props</fileIdRef>
               </source>
            </file>
         </addFile>
         <addFile>
            <file>
               <location>$(INSTALL_BASE)/$(PROPERTIES_DIR)</location>
               <name>def.properties</name>
               <source>
                  <fileIdRef>def_props</fileIdRef>
               </source>
            </file>
         </addFile>
      </actions>
   </actionGroup>
</action:artifact>
```

The uninstall artifact reverses the actions of installation by deleting files and removing directories.

Example 8-7 Uninstall artifact for ABC runtime

```
<?xml version="1.0" encoding="UTF-8"?>
<action:artifact
xmlns:action="http://www.ibm.com/namespaces/autonomic/solutioninstall/action"
xmlns:base="http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE"
xmlns:command="http://www.ibm.com/namespaces/autonomic/solutioninstall/command"
xmlns:iudd="http://www.ibm.com/namespaces/autonomic/solutioninstall/IUDD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:oca="http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions"
xsi:schemaLocation="http://www.ibm.com/namespaces/autonomic/solutioninstall/act
ion action.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/OsActions osActions.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/BASE base.xsd
http://www.ibm.com/namespaces/autonomic/solutioninstall/command command.xsd">
   <artifactType>Uninstall</artifactType>
```

```
<artifactSchemaVersion>1.2.0</artifactSchemaVersion>
<actionGroup xsi:type="oca:OsActionGroup">
   <actions>
      <removeFile>
         <location>$(INSTALL_BASE)</location>
         <name>env.bat</name>
      </removeFile>
      <removeFile>
         <location>$(INSTALL_BASE)/$(PROPERTIES_DIR)</location>
         <name>def.properties</name>
      </removeFile>
      <removeFile>
         <location>$(INSTALL_BASE)/$(PROPERTIES_DIR)</location>
         <name>abc.properties</name>
      </removeFile>
      <removeFile>
         <location>$(INSTALL_BASE)/$(LIB_DIR)</location>
         <name>xyz.jar</name>
      </removeFile>
      <removeFile>
         <location>$(INSTALL_BASE)/$(LIB_DIR)</location>
         <name>abc.jar</name>
      </removeFile>
      <removeFile>
         <location>$(INSTALL_BASE)/$(EXEC_DIR)</location>
         <name>script2.bat</name>
      </removeFile>
      <removeFile>
         <location>$(INSTALL_BASE)/$(EXEC_DIR)</location>
         <name>script1.bat</name>
      </removeFile>
      <removeFile>
         <location>$(INSTALL_BASE)/$(EXEC_DIR)</location>
         <name>abc.exe</name>
      </removeFile>
      <removeDirectory actionSetIdRef="osIBMBase">
         <location>$(INSTALL_BASE)</location>
         <name>$(PROPERTIES_DIR)</name>
      </removeDirectory>
      <removeDirectory actionSetIdRef="osIBMBase">
         <location>$(INSTALL_BASE)</location>
         <name>$(LIB_DIR)</name>
      </removeDirectory>
      <removeDirectory actionSetIdRef="osIBMBase">
         <location>$(INSTALL_BASE)</location>
         <name>$(EXEC_DIR)</name>
      </removeDirectory>
      <removeDirectory>
         <location>/</location>
```

```
            <name>$(INSTALL_BASE)</name>
         </removeDirectory>
      </actions>
   </actionGroup>
</action:artifact>
```

The <removeDirectory> element can contain a control attribute designating ‘all’ or ‘if-empty’. Use of this attribute could simplify the artifact and uninstall procedure.

Artifacts for optional features

We will not show the artifacts for the optional features. However, they are similar to those for the runtime component in that they create directories, copy files from the installation source, update text files, and so on. All of the files are available with the download package available for this book as described in Appendix A, “Additional material” on page 173.

8.3 Installation program

The installation program we will use is the same as the one we created in Chapter 6, “Creating an installation program” on page 93.

8.4 Packaging

Now that the installation program, manifest and payload files are ready, the next step is to bundle the package files (manifest and payload).

We bundle the payload and manifest files in a JAR named abc.jar. We can do this by executing the following command:

```
jar -cvf abc.jar FILES META-INF
```

This bundle is now self-sufficient with all the required files for the package.

To bundle the installation program along with the manifest and payload files, we will create an additional bundle that will include the installation program and related files. This new bundle is called abc-bundle.jar and is created with the following command:

```
jar -cvf abc-bundle.jar com Response.properties abc.jar
```

8.5 Running the sample

Our very simple installation program does not provide the option to pass in an archive file as a parameter and automatically extract the manifest and payload files from it. So, to see our sample in action, we need to extract the bundles that we described in the previous section. The following steps assume a Windows environment. Similar steps can be carried out in a Linux environment with minor changes in syntax.

To unzip and test our example, perform the following steps:

1. Create a temp directory, say C:\abc-inst-src.
2. Copy the abc-bundle.zip to C:\abc-inst-src.
3. Extract the files from abc-bundle.jar as follows:

   ```
   jar xvf abc-bundle.jar
   ```

   This will result in two files and a directory structure being created (Response.properties, abc.jar and ./com). The com directory contains the directory structure, source file, and class file for the SampleInstaller installation program.
4. Extract the manifest and payload files by using the **jar** command as follows:

   ```
   jar xvf abc.jar
   ```

   This creates and populates the ./FILES and ./META-INF directory structure containing the payload and manifest files, respectively.
5. Ensure that the CLASSPATH environment includes the current directory:

   ```
   set CLASSPATH=.;%classpath%
   ```

The environment is now ready to test our installation. Before proceeding, let us take a look at our response file (Response.properties), as shown below.

Example 8-8 Response.properties file for ABC example

```
Variable#SoftwareIUTypeID[11111111111111111111111111111111,1.0,RootIUTypeID[123412341234123412
34123412341234,1.1]]#INSTALL_BASE=/ABC_APP
Feature#RootIUTypeID[1234123412341234123412341234123412341234,1.1]#Feature_Documentation
Feature#RootIUTypeID[1234123412341234123412341234123412341234,1.1]#Feature_DK
Feature#RootIUTypeID[1234123412341234123412341234123412341234,1.1]#Feature_Samples
```

This response file will set the variable INSTALL_BASE to ABC_APP. This is the directory used as the base for the installation of the ABC package. If you look in the deployment descriptor, you can see that the default value for this variable is /myapps/ABC_APP.

The three optional features are also installed by default with this response file. You can comment out any or all of the lines in this file to use the default variable value and choose the features that are or are not installed.

Now we are ready to execute the installation. The command we use and the results of the execution are shown below. Be sure that the Solution Installation setenv script has been executed previously to executing the command below.

Example 8-9 Installation of the ABC sample application

```
C:\abc-inst-src>java com.itso.ac.si.SampleInstaller Meta-INF/packagedIU.xml
Response.properties create
****** Sample Installer ******
Simple program to demonstrate
some features of the
Solution Installation for
Autonomic Computing toolkit
******************************


Check name:Windows_Type_Check
 Check status: true

Property name : OSType
Property value : Windows XP

Check name:Linux_Check
 Check status: false

Check name:Windows_Version_Check
 Check status: true

Property name : version
Property value : 5.1.Service Pack 1.Build 2600

Check name:Linux_Version_Check
 Check status: false

Check name:Diskspace_Check
 Check status: true

Property name : availableSpace
Property value : 47737712

Parameter Overrides:
 Variable : INSTALL_BASE
 Value : /ABC_APP
```

```
FeatureId : Feature_Samples
FeatureId : Feature_DK
FeatureId : Feature_Documentation
Hosting Environment found is:
mrid:http://w3.ibm.com/namespaces/2003/OS_componen
tTypes:Operating_System

Begin Create Operation
SampleInstaller has recieved a changeOccurred Event: SUBMITTED
SampleInstaller has recieved a changeOccurred Event: EXECUTING
SampleInstaller has recieved a changeOccurred Event: EXECUTING
SampleInstaller has recieved a changeOccurred Event: EXECUTING
SampleInstaller has recieved a changeOccurred Event: EXECUTING
SampleInstaller has recieved a changeCompleted Event: COMPLETED
```

From the above output, you can see the results of various checks that were executed based on the checks requested in our deployment descriptor. The override for the INSTALL_BASE parameter is shown and the list of selected features are shown. Finally, the create operation is executed and is completed without error. The result of a **listiu** command executed after this installation is shown below.

Example 8-10 Result of listiu command

```
C:\abc-inst-src>listiu
IU      UUID: 11111111111111111111111111111111  Name: ABC_Runtime_SIU
Version: 1.0
IU      UUID: 22222222222222222222222222222222  Name: ABC_Documentation_SIU
Version: 1.0
IU      UUID: 33333333333333333333333333333333  Name: ABC_Samples_SIU
Version: 1.0
IU      UUID: 44444444444444444444444444444444  Name: ABC_DK_SIU
Version: 1.0
IU      RootIU UUID: 12341234123412341234123412341234   Name:
ABC_Sample_Package Version: 1.1
Feature (RootIU UUID: 12341234123412341234123412341234  Name:
Feature_Samples)Feature Name:Feature_Samples
Feature (RootIU UUID: 12341234123412341234123412341234  Name:
Feature_Documentation)Feature Name:Feature_Documentation
Feature (RootIU UUID: 12341234123412341234123412341234  Name:
Feature_DK)Feature Name:Feature_DK
```

The package may be uninstalled in a similar fashion as install. Only the operation requested on the command line would change. The command for uninstalling the ABC package is:

```
java com.itso.ac.si.ABCInstaller META-INF\packagedIU.xml Response.properties
delete
```

We will leave the execution of this to the reader.

8.6 Summary

This chapter has described a very simple application and how to create the descriptor and artifact files required to install (and uninstall) it. Though very basic, it provides an example that the reader can build on to test more advanced examples and explore the facilities provided by the Solution Installation toolkit.

A

Additional material

This publication refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this publication is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG247099

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG247099.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
samp7099.jar	An archive file containing various sample code, descriptors and other artifacts

System requirements for downloading the Web material

The materials included in the above jar file are relatively small text files and Java source files. There are no extraordinary requirements for downloading and viewing the files contained within the zip file. To compile the Java source and test the scenarios described in this publication, a system with the Solution Installation toolkit and appropriate Java environment is required.

How to use the Web material

Create a subdirectory (folder) on your workstation, and copy the samp7099.jar file to it. Extract the files by issuing the following command:

```
jar xvf samp7099.jar
```

This will result in the creation of four new jar files:

- SampleInstaller.jar
- CH6-Example1.jar
- CH6-Example2.jar
- CH8-Example.jar

Again, extract the files from each of these jar files by using the commands:

```
jar xvf SampleInstaller.jar
jar xvf CH6-Example1.jar
jar xvf CH6-Example2.jar
jar xvf CH8-Example.jar
```

By extracting these files, individual directories will be created with the source code for the SampleInstaller program described in Chapter 6, “Creating an installation program” on page 93, and the descriptors and artifacts for the various examples described in the related chapters.

Please refer to the individual chapters for information about the sample files.

Since the same SampleInstaller program is used with all of the examples, you can set your environment such that the directory where the SampleInstaller.jar file is extracted to is in your CLASSPATH environment variable. Then you will be

able to use that program from within each of the other sample directories that contain the descriptor and artifact files. For example, let us assume that you extracted all of the above files in a directory named MySITest. The following directory structure will be created:

```
\MySITest
   |- com
   |  |-itso
   |     |-ac
   |        |-si
   |           SampleInstaller.java
   |           SampleInstaller.class
   |-SIsample
   |  |-FILES
   |  |-Meta-INF
   |  .
   |  .
   |-SISample2
   |  .
   |  .
   |-SISample3
      .
      .
```

Set your CLASSSPATH to include \MySITest. For instance, in a Windows environment issue the command:

```
set CLASSPATH=C:\MySITest;%CLASSPATH%
```

Now, from each of the sample directories, you can execute commands as described in the individual chapters, such as:

```
c:\MySITest\SIsample > java com.itso.ac.si.SampleInstaller
./Meta-INF/packagedIU.xml Response.properties create
```

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 177. Note that some of the documents referenced here may be available in softcopy only.

- *A Practical Guide to the IBM Autonomic Computing Toolkit,* SG24-6635

Online resources

These Web sites and URLs are also relevant as further information sources:

- IBM Autonomic Computing

  http://www.ibm.com/autonomic/

- IBM Autonomic Computing blueprint

  http://www.ibm.com/autonomic/pdfs/ACwpFinal.pdf

- IBM developerWorks® autonomic computing page

  http://www.ibm.com/developerWorks/autonomic

- XML technical report

  http://www.w3.org/TR/2000/REC-xml-20001006

- XML schema technical report

  http://www.w3.org/TR/2001/REC-xmlschema-0-20010502

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

T

U

V

W

X

Z

A First Look at Solution Installation for Autonomic Computing

Enable autonomic solution installation

Learn about the key technologies

Get started today

This IBM Redbook provides a first look at the Solution Installation capability that is a key component of the IBM Autonomic Computing initiative. Autonomic computing technologies enable complex computing systems to be self-configuring, self-healing, self-optimizing, and self-protecting. One aspect of this is the ability to automate the change management processes associated with deploying and managing software required for business solutions.

The Solution Installation technologies that are described are evolving and will be enhanced over time. This publication provides the reader with an opportunity to understand the underlying concepts and facilities. This publication is based on Version 1.2 of the Solution Installation toolkit and describes both architectural concepts as well as specific examples that apply to the current version. We attempt to make clear when a key architectural concept is not fully implemented at this time.

Readers of this book will gain an understanding of the basic concepts, and how the architecture described will provide business value as it becomes fully implemented and utilized by IBM and other software vendors.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7099-00 ISBN 073849030X